

Logarithmic Communication for Distributed Optimization in Multi-Agent Systems

PALMA LONDON, California Institute of Technology, USA
SHAI VARDI, Purdue University, USA
ADAM WIERMAN, California Institute of Technology, USA

Classically, the design of multi-agent systems is approached using techniques from distributed optimization such as dual descent and consensus algorithms. Such algorithms depend on convergence to global consensus before any individual agent can determine its local action. This leads to challenges with respect to communication overhead and robustness, and improving algorithms with respect to these measures has been a focus of the community for decades.

This paper presents a new approach for multi-agent system design based on ideas from the emerging field of local computation algorithms. The framework we develop, LLocal Convex Optimization (LOCO), is the first local computation algorithm for convex optimization problems and can be applied in a wide-variety of settings. We demonstrate the generality of the framework via applications to Network Utility Maximization (NUM) and the distributed training of Support Vector Machines (SVMs), providing numerical results illustrating the improvement compared to classical distributed optimization approaches in each case.

CCS Concepts: • **Theory of computation** → **Convex optimization**; • **Computing methodologies** → **Multi-agent systems**; *Distributed algorithms*.

Additional Key Words and Phrases: distributed algorithms; distributed optimization; multi-agent systems

ACM Reference Format:

Palma London, Shai Vardi, and Adam Wierman. 2019. Logarithmic Communication for Distributed Optimization in Multi-Agent Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 3, Article 48 (December 2019), 29 pages. <https://doi.org/10.1145/3366696>

1 INTRODUCTION

This paper introduces a novel approach for distributed optimization in multi-agent systems based on ideas from an emerging area in theoretical computer science – *local computation algorithms* [75] – that allows distributed agents to compute a local action or estimate with exponentially reduced communication and significantly improved robustness in sparse settings.

Distributed optimization is an area of crucial importance to the design and control of multi-agent systems. It provides a framework for the design of multi-agent systems where the system goal is formalized via a global objective and the distributed agents work together to solve this global optimization problem. Then, the agents determine their action by looking at the piece of the global solution associated with them. Crucially, in this framework an agent’s goal is to determine its own action, i.e., its piece of the global solution. *It does not necessarily need to know the full global solution.*

Authors’ addresses: Palma London, plondon@caltech.edu, California Institute of Technology, 1200 E. California Blvd., Pasadena, California, 91125, USA; Shai Vardi, svardi@purdue.edu, Purdue University, 403 W State St, West Lafayette, Indiana, 47907, USA; Adam Wierman, California Institute of Technology, USA, adamw@caltech.edu.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

2476-1249/2019/12-ART48 \$15.00

<https://doi.org/10.1145/3366696>

Settings where distributed optimization has been used in the design of multi-agent systems are numerous and varied. Examples include management of content distribution networks and data centers [12, 65], communication network protocol design [40, 51, 83], trajectory optimization [34, 44], formation control of vehicles [71, 85], sensor networks [49, 64], control of power systems [25, 70], and management of electric vehicles and distributed storage devices [30]. Further, recently such approaches have become prominent in the emerging field of federated machine learning [42, 56], where data is distributed across a set of agents and the goal of the agents is to train a model using the full data set without sharing data between them.

Distributed optimization is a field with a long history. Beginning in the 1960s approaches emerged for solving large scale linear programs via decomposition into pieces that could be solved in a distributed manner. For example, two early approaches are Bender's decomposition [8] and the Dantzig-Wolfe decomposition [22, 23], which can both be generalized to nonlinear objectives via the subgradient method [9, 60, 82]. Today, there are a wide variety of approaches for distributed optimization in use, e.g., primal decomposition [9, 45], dual decomposition [13, 37, 40, 50, 60, 83, 86], subgradient methods [59, 62], and proximal gradient descent methods [81], to name a few. Often these methods employ consensus schemes as a mechanism for distributing the computation among the processing units, forming the basis for many first order and second order distributed optimization algorithms, e.g., [11, 61].

Despite the wide variety of approaches to distributed optimization in multi-agent systems, the approaches that are studied and used today are similar at a high level – and this similarity leads to fundamental limitations on their scalability and robustness. In particular, all the approaches listed above, at their core, pass current estimates of the global solution between agents in a sequential process, gradually improving those estimates at each step with the goal of convergence to a (near) optimal solution, i.e., consensus. Classically, in such approaches, the distributed agents are required to store, update, and broadcast a vector of dimension that matches that of the full system-wide solution to the problem at each step, which for multi-agent systems in modern applications can be enormous. Further, no individual agent can determine its own action or estimate without global convergence of all agents in the network. This is a result of the fact that distributed optimization algorithms are designed to allow each distributed agent to compute the full global solution. But, this is overkill for multi-agent systems, where typically an agent needs only to compute its local piece of the solution in order to determine its action.

As a result, there are a number of serious and fundamental challenges when it comes to applying distributed optimization algorithms in the design of multi-agent distributed systems.

First, since the network size can be enormous, consisting of tens or hundreds of thousands of distributed agents (for example, in emerging internet of things (IoT) applications, the communication and storage demands for each iteration may be extreme. In fact, in most such approaches, e.g., consensus-style approaches, the communication within a single round requires $O(n)$ messages, typically containing a current estimate of the global solution. There has been considerable research that seeks to reduce the communication overhead of these approaches, e.g., [36, 59, 62, 81]. These approaches seek to partition the global solution into multiple blocks, each of which can be communicated less frequently, thus lowering the communication overhead. However, to this point, order-of-magnitude improvements have not been found for general classes of optimization problems.

Second, the iterative convergence of traditional distributed optimization algorithms means that the convergence of *all nodes* can be delayed if a single node or communication link is congested. For example, if there is communication lag in one part of the network, a consensus algorithm cannot reach consensus, and thus no agent in the network can determine its local action. Such "stragglers" are frequent in modern distributed systems and lead to significant delays in many

distributed optimization designs. The importance of this issue has been recognized for decades, and there has been considerable work toward developing asynchronous approaches for dual descent and consensus algorithms, e.g., [7, 16, 87, 91]. However, even asynchronous algorithms require all nodes to communicate repeatedly in order for consensus to be achieved. Thus, if a set of agents is suffering from poor communication conditions, agents across the network must still wait for that part of the network to converge in order to determine their actions.

Third, classical approaches result in designs where any changes in network structure due to communication links failing or agents entering/leaving the network means that the algorithm is brought to a halt and needs to restart the convergence process. Again, this is a long-standing issue and the design of fault tolerant distributed optimization has received considerable attention. Robustness to failures and changes in the system are typically addressed through the design of fault-tolerant, Byzantine distributed optimization approaches, e.g., [18], however such approaches require significant adjustments to the classical algorithms and come at significant expense in terms of convergence rates and optimality guarantees.

Fourth, because classical distributed algorithms require *global* convergence/consensus before any individual agent can determine its local action, a single agent computing its individual action or estimate imposes communication and computation demands on every agent in the network. This introduces unnecessary overhead and delay since it means that an individual agent is impacted by stragglers, agents entering/exiting, etc., across the *whole system* even though it only seeks to compute its *local* action. Ideally, an agent would be able to compute its part of the solution without the need to compute the full global solution.

Goal. *In this paper, we seek to develop a new approach for distributed optimization in multi-agent systems that can reduce the communication overhead of traditional approaches, while also guaranteeing robustness to communication delay and failures in the system. To accomplish this, we seek a design that allows an individual agent to compute its local optimal action without the need for global communication.*

Our approach toward achieving this goal is to develop a novel connection between distributed optimization and an emerging sub-field of theoretical computer called *local computation algorithms* (LCAs) [75] – applying local computation algorithms to optimization problems for the first time. The LCA framework was formally introduced by [75] in order to connect a variety of algorithms with similar goals that had recently appeared in distinct areas [5, 39, 76]. Until our work, the field has focused on the design of LCAs for graph problems such as matching, maximal independent set, and coloring [3, 27, 46, 73]. In this paper we show that the approach is promising for distributed optimization as well.

The defining property of local computation algorithms is that they seek to compute a local “piece” of the solution to some algorithmic problem using only information that is “close” to that piece of the problem. For example, an LCA for matching allows each node in the graph to compute its own match locally by communicating only with a small neighborhood of other nodes, without computing the entire matching for the graph. Yet, if all nodes run the LCA, then the solution each node computes is part of the same global matching.

In the context of distributed optimization in multi-agent systems, this means that when running an LCA, a distributed agent computes its own action or estimate (its local piece of the solution to the global optimization problem) *without computing or communicating the global solution*. However, if every agent runs the LCA, then the agents together (approximately) solve the global optimization problem, i.e., compute pieces of the same global solution. So, if there exists an LCA for the optimization problems used in networked and distributed systems, it would allow an agent to compute its local action without waiting for global consensus to be achieved. Thus, it could provide a significant reduction in communication compared to traditional approaches while also improving robustness

to stragglers and agents entering/exiting the system, since stragglers and agents entering/exiting would only impact an agent's computation of their action if they happen within the small, local neighborhood of the agent.

Contributions. In this paper we develop the first local computation algorithm for convex optimization, LOCO (Local Convex Optimization). This optimization framework represents a fundamentally new approach for distributed optimization in multi-agent systems that allows an individual agent to compute its action with *exponentially less communication* than traditional approaches, while maintaining robustness to both stragglers and the entrance/exit of agents into the system. Further, LOCO allows an individual agent to compute its action or estimate without the need for global convergence, and thus without the need for global communication and computation.

Concretely, we consider a multi-agent system with N distributed agents that wish to compute actions or estimates $x_j \in \mathbb{R}^{q_j}$, where q_j is the dimension of the actions for agent j , so that the combination of the actions forms a global solution $x \in \mathbb{R}^n$ to a constrained optimization problem of the following form. This form is of interest for a wide variety of problems in multi-agent networked systems, e.g., regression problems and support vector machines [21, 35, 38], distributed inference in sensor networks, which has broad applications to the Internet of Things [32, 33, 64, 68], inference in graphical models [2, 72], relaxations of maximum a posteriori (MAP) estimation problems [78], network utility maximization (NUM) problems, [40, 50], management of content distribution networks and data centers [12, 65], and control of power systems [25, 70]:

$$\begin{aligned} & \text{minimize} && \sum_{j=1}^N f_j(x_j) && (1) \\ & \text{subject to} && \sum_{j=1}^N a_{ij}x_j \geq b_i && i \in [m] \\ & && x \geq 0 && \end{aligned}$$

where $x_j \in \mathbb{R}^{q_j}$ and $f_j : \mathbb{R}^{q_j} \rightarrow \mathbb{R}$ are convex functions. We allow overlap or coupling between the functions f_j ; i.e., a component of the entire solution $x \in \mathbb{R}^n$ may appear in multiple local functions f_j . When this happens, the agents' actions are coupled through the overlapping variables. Formally, this implies that $\sum_{j=1}^N q_j \geq n$, where if there is equality there are no variables that appear in multiple agents' actions, x_j , and if the inequality is strict there are variables that appear in two or more agents' actions. Additionally, $a_{ij} \in \mathbb{R}^{\leq q_j}$ are submatrices of an $A \in \mathbb{R}^{m \times n}$ matrix, where $[a_{i1}, \dots, a_{ij}, \dots, a_{in}]$ is the i th row of A , and $b \in \mathbb{R}^m$.

The problem is defined over a network, where each agent j is associated with a node j , a variable x_j , and a function f_j . The problem data (m constraints), A and b , are distributed over the agents, and the N agents are completely distributed. In this paper, we are concerned with settings in which n , m , and N are large, but each local function f_j depends on a relatively small number of components of x , i.e., the dimension of the agents' actions is small, and the matrix A is sparse (has a small number of non-zero entries in each row and column).

We would like to emphasize that the task for an individual, distributed agent is to compute its own local action or estimate, x_j . The agent does not need the full global solution x , only its local piece. Note that in traditional approaches for distributed optimization, e.g., consensus and dual descent, a byproduct of the algorithms used is that each agent computes the full global solution x , which may be of significant size and requires global convergence (and thus communication and computation by every agent in the network) to compute. This should not be viewed as a *feature* of these algorithms, instead it is an unnecessary *overhead* in the case of multi-agent systems (since the agent is only responsible for its local action).

A key insight in the design of LOCO is that is not necessary for an individual agent to compute the global solution in order to determine its individual action. Instead, *it is possible for an agent to compute its local "piece" of the solution x_j without computing the full global solution x* . To achieve this,

the fundamental idea of LOCO is to, for a given distributed agent, define a *local problem* associated with the agent's action (variable) x_j , which is defined on a subset X_j of the primal variables and a subset Y_j of the data (constraints). The agent j then solves its local problem using a given algorithm that is purely local. This produces the local action/estimate, x_j , that is a piece of an (approximately) optimal solution to the global problem x . Further, if every distributed agent runs the same local algorithm, then x is computed.

Note that the sets X_j and Y_j used by LOCO are much smaller in size than n and m respectively (the dimensions of the original problem), resulting in a dramatic dimension reduction and thus a reduction in communication and computation when the matrix A is sparse. We show (Theorem 3) that, when the data matrix A is sparse, i.e., the maximum number of non-zero entries in a row or column is bounded by a constant, X_j and Y_j both have sizes on the order of $O(\log m)$. We utilize this to guarantee that a small number of messages needs to be passed, and that the messages passed are small in size.

More generally, we provide worst-case guarantees on the performance of LOCO with respect to the amount of communication it requires and the quality of the solution. Regarding communication, the process of determining sets X_j and Y_j requires $O(\log m)$ messages with high probability. After this step, solving the local problems at each node requires no communication. Note that this is an exponential reduction compared to the $O(n)$ communication required during *each round* of traditional approaches such as consensus and dual descent when A is sparse.

We also provide worst-case guarantees on the performance of LOCO with respect to the quality of the solution. Since the nodes do not have access to the entire problem under LOCO, it is unreasonable to expect an exact solution. Instead, LOCO produces a feasible, α -approximation of the optimal solution, where α depends on the given algorithm \mathcal{A} used to solve the local problem of an agent (Theorem 3). Our numeric results in Section 5 highlight that the approximation error of LOCO matches that of ADMM in many cases, while using orders-of-magnitude less communication.

To develop algorithms to solve the local problem of an agent, we prove a reduction that allows generic online algorithms to be "converted" into local optimization algorithms. This approach is based on an insight in a foundational result in the local computation literature, which shows that online algorithms can be converted into local algorithms with the same performance guarantee in *graph problems* with bounded degree [54, 73]. Our contribution is to, for the first time, show that a similar reduction is possible for *optimization problems*, where the bounded degree property is replaced by the sparsity of the constraint matrix. This enables us to prove that if an online algorithm, \mathcal{A} , running on global information is guaranteed to output an α -approximate solution, then when LOCO uses the algorithm to compute the local solution of an agent the resulting solution is also an α -approximation. Thus, the LOCO framework inherits the approximation ratio of \mathcal{A} .

To illustrate the power of this reduction and the generality of LOCO, we provide specific results for two different classes of optimization problems of significant practical interest (Corollaries 4 and 5). These two results use two different online algorithms as the algorithm for solving the local problem of an individual agent in LOCO; thus highlighting the generality of the LOCO framework. Specifically, we show that LOCO achieves an $O(\log m)$ -approximation in the case that the objective functions are linear and an ϵ -approximation in the case of linear SVM problems. Beyond these theoretical guarantees, we also provide numerical case studies for these two examples in Section 5. The case studies show order-of-magnitude improvements in communication time are possible using LOCO, and that this is possible without incurring excessive approximation error.

2 PROBLEM FORMULATION

We study a multi-agent system with N distributed agents and no central control. The agents may communicate with neighbors, but communication with a centralized processing unit is prohibited.

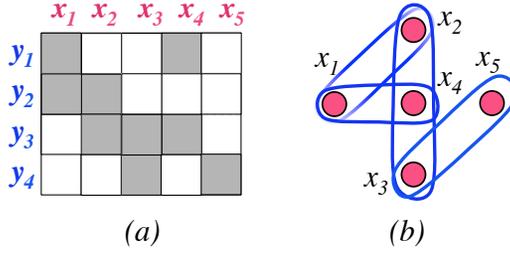


Fig. 1. The constraint matrix is depicted in (a), and the *hypergraph* \mathcal{H} in (b). Red shaded nodes represent the primal variables and blue hyperedges represent the constraints, or dual variables. Hyperedges encircle primal variables which appear together in a constraint.

The system is designed such that the agents seek to compute their local actions or estimates x_j for each of the N agents and that the set of all agent actions solves a constrained convex optimization problem with a separable objective and coupled constraints. Specifically, the agents together seek to solve an optimization problem of form (1).

The problem is defined over a network G , and each agent j is associated with a node j , a variable x_j , and a function f_j . The problem data (constraints), A and b , are distributed over the agents. Each agent has a subset of the constraints, or rows of the A matrix. There may be copies of the same constraints at different nodes.

In our algorithm, the problem is represented as a hypergraph $\mathcal{H} = (V, H)$. The set of nodes in the hypergraph $V = \{1 \dots N\}$ is the same as that of G , where each node corresponds to a variable x_j . Hyperedges $H = \{1 \dots m\}$ correspond to constraints. We associate each constraint with a dual variable $y_i \forall i \in [m]$, and refer to primal constraints and dual variables interchangeably. As an example, in Figure 1, nodes encircled by a hyperedge correspond to primal variables that appear together in a constraint.

We measure the performance of an algorithm in this setting with respect to the amount of communication it requires and the quality of the solution it produces. To measure the amount of communication, we define a *message* to be information that is sent between neighbors in a graph and we define *message complexity* to be the number of messages sent across edges in order to compute the solution. In our setting, small pieces of the constraint matrix, A , are passed between nodes. Since the A matrix is very sparse, this amounts to only sending several matrix coefficients a_{ij} at a time, along with their index information (i, j) , and the coefficient b_i . We define a message with respect to each i th constraint to be the list of matrix coefficients $\{a_{ij} \forall j \in [n] : a_{ij} \neq 0\}$, for a given i th row of the matrix A , along with the coefficient b_i .

When the algorithm uses randomization, we prove bounds on the message complexity that hold with probability at least $1 - \frac{1}{m^\gamma}$, where m is the number of constraints and $\gamma > 0$ can be an arbitrarily large constant. We denote this by $1 - \frac{1}{\text{poly } m}$. We do not bound the size of the messages, but note that in both our algorithm and most dual descent and consensus algorithms the message lengths are of order $O(\log(n + m))$.

By default, the graph we consider communication over is the hypergraph \mathcal{H} . However, we can also describe communication with respect to the physical network G . The difference between these is a function of the *sparsity* of A , which we define as $d = \max\{d_r, d_c\}$, where d_r and d_c denote the maximum number of nonzero entries in rows and columns of A respectively. We say that A is *sparse* if the sparsity of A is bounded by a constant. Thus, given that the constraint matrix A has sparsity d , the number of messages required on G compared to \mathcal{H} differs by a factor of at most d^2 .

To measure the quality of the solution of an algorithm in this setting we use the *approximation ratio*. An algorithm is said to produce an α -approximate solution if its solution is guaranteed to be at most αOPT , where OPT is the value of the optimal solution. In our empirical results, we compare the performance of LOCO to the dual decomposition method ADMM, for which approximation ratio is not a standard measure. Thus, empirical comparisons are made using *relative error*, defined in Section 5.1.1, which is related to, but different from, the approximation ratio.

This setting and the performance measures we use are of broad interest in multi-agent systems. For example, the setting has been considered in regression problems and support vector machines [21, 35, 38], distributed inference in sensor networks, which has broad applications to the Internet of Things [32, 33, 64, 68], inference in graphical models [2, 72], relaxations of maximum a posteriori (MAP) estimation problems [78], Network Utility Maximization (NUM) problems, [40, 50], management of content distribution networks and data centers [12, 65], and control of power systems [25, 70]:

In this paper, we use two examples to highlight the generality of the LOCO framework: NUM and SVM, which we describe in Subsections 2.1 and 2.2 respectively. With the example of SVM, we also highlight the potential for LOCO to be used in settings that are not fully distributed.

2.1 Network Utility Maximization (NUM)

To illustrate the application of LOCO to multi-agent systems, we focus on the example of NUM, which is a general class of optimization problems that has seen widespread applications in multi-agent systems, from the design of TCP congestion control [40, 50, 51, 83] to understanding of protocol layering as optimization decomposition [19, 66] and power system demand response [48, 77]. For a recent survey on NUM see [90].

The NUM framework considers a network containing a set of sources (agents) $\mathcal{S} = \{1, \dots, m\}$ and links $\mathcal{L} = \{1, \dots, n\}$ of capacity c_j , for $j \in \mathcal{L}$. Source $i \in \mathcal{S}$ is characterized by $(L_i, f_i, \underline{x}_i, \bar{x}_i)$: $L_i \subseteq \mathcal{L}$ is a path in the network; $f_i : \mathbb{R}_+ \rightarrow \mathbb{R}$ is a concave utility function; \underline{x}_i and \bar{x}_i are the minimum and maximum transmission rates of source i respectively.

The goal of a source is to determine its rate x_i such that the aggregate utility of all sources is maximized. Source i attains a concave utility $f_i(x_i)$ when it transmits at rate x_i along path L_i , within the minimum and maximum rates allowed. The maximization of aggregate utility is formulated as

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^m f_i(x_i) && (2) \\ & \text{subject to} && A^T x \leq c \\ & && \underline{x} \leq x \leq \bar{x}, \end{aligned}$$

where $A \in \mathbb{R}_+^{m \times n}$ is defined as $A_{ij} = 1$ if $j \in L_i$ and 0 otherwise.

Different choices of f_i correspond to different network goals. Some of the most common in networking settings are (i) setting $f_i(x_i) = x_i$ to maximize throughput; (ii) setting $f_i(x_i) = \log(x_i)$ to achieve proportional fairness; (iii) setting $f_i(x_i) = -1/x_i$ to minimize potential delay [50, 55].

While consensus and dual descent methods have received considerable attention in the NUM literature, note that sources *do not* need to know the global solution. They only need to know their local rate, x_i . Thus, NUM is a natural application where local computation can provide significantly reduced communication and improved robustness by eliminating the demand that every agent converge to the full, global solution.

We use numerics in Section 5 to show the improvements LOCO provides compared to classical approaches for NUM. In these examples, we focus on $f_i(x_i) = x_i$, i.e., maximizing throughput, since it is typically viewed as the most challenging. However, the LOCO framework can be applied to any NUM objective.

2.2 Support vector machines (SVMs)

Federated machine learning is an increasingly prominent framework that seeks to train machine learning models in settings where data is distributed among multiple agents due to privacy concerns. This approach has received significant attention from researchers in recent years, e.g., [42], and appears in industry as well, e.g., [56]. Inspired by this, our second example illustrates how LOCO can be used for distributed training of an SVM.

SVMs represent a core model in machine learning that is crucial for applications in both regression and classification. While there are many variations of SVMs, we use the following classical version to illustrate the application of LOCO. We consider the task of fitting an SVM to data pairs $S = \{(z_i, y_i)\}_{i=1}^m$, where $z_i \in \mathbb{R}^n$ and $y_i \in \{+1, -1\}$ is a label for each data pair. Traditionally, this problem is presented as a regularized optimization problem of the following form:

$$\text{minimize}_x \sum_{(z_i, y_i) \in S} \max\{0, 1 - y(x^T z)\} + \lambda \|x\|_2^2. \quad (3)$$

As stated the above optimization does not match the form of (1), however there are a number of standard transformations that lead to matching forms. For example, we use the case of linear SVMs to illustrate LOCO. For linear SVMs, (3) can be written in the following form [38], which matches (1):

$$\begin{aligned} & \text{minimize}_{x, \xi_i \geq 0} \quad \frac{1}{m} \sum_{i=1}^m \xi_i + \lambda \|x\|_2^2 & (4) \\ & \text{subject to} \quad y_i(x^T z_i) \geq 1 - \xi_i, \quad \forall i \in [m]. \end{aligned}$$

Here, the local variables associated with the agents are ξ_i , and these can be computed in a completely distributed way using the LOCO framework, see Section 5 for experiments demonstrating the performance and robustness improvements of this approach.

This application highlights another point about LOCO. It can be applied in both distributed and parallel settings. In particular, if the goal is to determine the whole global solution, i.e., the full SVM model, then one simple “join” step where each agent sends the solution to a central entity accomplishes this. Thus, LOCO can be used to provide a parallel SVM implementation that is robust to stragglers and failures of compute nodes.

3 A LOCAL OPTIMIZATION FRAMEWORK

In this section we introduce the framework that is the main contribution of the paper: *Local Convex Optimization* (LOCO). We describe the framework and give intuition for it in this section and then, in the next section, we focus on providing provable guarantees on communication and accuracy.

LOCO consists of two steps. In the first, LOCO generates a (small) localized neighborhood for each variable or source. In the second, LOCO simulates an online algorithm on the localized neighborhood. Note that the first step is independent of the online algorithm, and the second is independent of the method used to generate the localized neighborhoods. Therefore, one should think of LOCO as a general framework that can yield a variety of algorithms for different classes of optimization problems depending on the online algorithm it is instantiated with. For example, we can use different online algorithms for the second step of LOCO depending on whether we consider NUM or SVM, as we do in the next section.

More specifically, the details of the two follow and are summarized in Algorithm 1 below.

Step 1: Set up the Local Problems. For each agent $j \in V$, define an associated *local problem*, consisting of a subset X_j of the primal variables and a subset Y_j of the constraints, or dual variables.

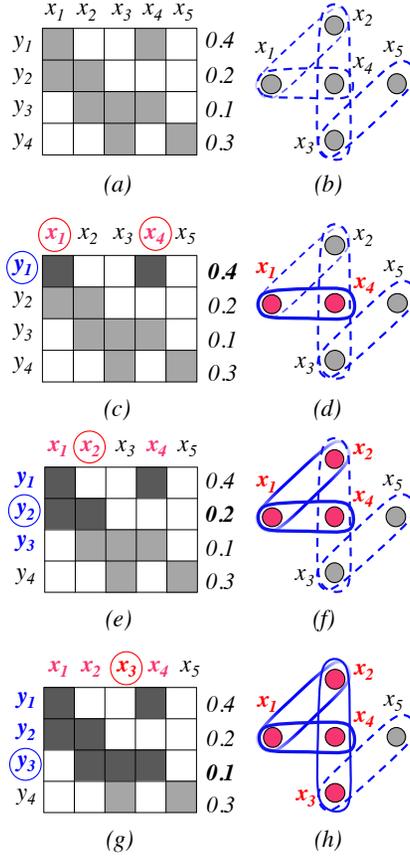


Fig. 2. An illustration of LOCO. The constraint matrix A is depicted in (a), where shaded entries represent non-zeros. The rankings of the constraints are indicated next to their corresponding rows in A . The hypergraph \mathcal{H} is depicted in (b). Figures (c)-(h) illustrate the construction of sets X_1 and Y_1 for the local problem associated with variable x_1 . The darkest shaded matrix elements in (c), (e), and (g) indicate constraints as they are received by agent 1. Blue emboldened hyperedges in (d), (f), and (h) represent constraints added to Y_1 . Red shaded nodes represent variables added to set X_1 . The process stops in (g) because rankings $0.1 < 0.3$. The local problem associated with variable x_1 is defined on variables $X_1 = \{1, 2, 3, 4\}$ and constraints, or dual variables, $Y_1 = \{1, 2, 3\}$.

The local problem is of the form:

$$\begin{aligned}
 & \text{minimize} && \sum_{k \in X_j} f_k(x_k) && (5) \\
 & \text{subject to} && \sum_{k \in X_j} a_{ij} x_k \geq b_i && i \in Y_j \\
 & && x_k \geq 0 && k \in X_j
 \end{aligned}$$

In order to construct sets X_j and Y_j , first generate a random ordering on the constraints. Let $r : [m] \rightarrow [0, 1]$ be a function that assigns each constraint, or dual variable y_i , a real number between 0 and 1 uniformly at random. We call $r(i)$ y_i 's rank. For more about generating r efficiently, see Appendix C. We assume that all of the nodes have access to r , and hence can compute the rank of any constraint.

Construct X_j and Y_j as follows. At node j , calculate the rank of each of the constraints in which variable x_j appears, i.e., for all i such that $a_{ij} \neq 0$. Among these constraints, identify the index of the highest ranked constraint: $h = \operatorname{argmax}_j \{r(i) | a_{ij} \neq 0\}$. Add constraint h to set Y_j . In a recursive fashion, at a given node j' , contact j' 's neighbors in \mathcal{H} to learn which constraints they appear in: $i \in [m]$ s.t. $a_{ij'} \neq 0$. At node j' , calculate the ranks of each of these constraints. Add to Y_j the constraints that have lower rank than the constraint most recently added to Y_j , i.e., $r(i) < r(h)$, and add to X_j the primal variables that appear in those constraints. Repeat this process until all visited neighbors appear in constraints that have higher rank than the last constraint added to Y_j . This process is stated concretely in Algorithm 1; see Figure 2 for an example.

Step 2: Solve the Local Problems. The j th agent solves the j th local problem (5) using any existing convex optimization algorithm that is a local sequential algorithm in the following sense.

DEFINITION 1. A local sequential algorithm for problems of form (1) is one that observes input sequentially. Assume that the constraints arrives according to some order π , for simplicity, we set $\pi(i) = i$; that is, the constraint associated with the dual variable y_i arrives at step $i \in [m]$. At step $i = 1, \dots, m$, only y_i and x_j such that $a_{ij} \neq 0$ are possibly updated, and their new values depend only on the value (at step i) of primal variables x_j such that $a_{ij} \neq 0$.

Note that all the updates the local sequential algorithm makes at step i are based only on the values of $x_j \forall j \in V$ for which $a_{ij} \neq 0$ when y_i arrives. Local sequential algorithms include most online algorithms, such as the algorithms in [14] for covering or packing linear programs; those in [6] for convex covering and packing problems with linear constraints; and in [24] for general convex conic covering problems. For example, NUM is a packing problem with linear constraints, and thus LOCO can be run with the algorithms of [14] or [6]. Local sequential algorithms also include many stochastic gradient descent methods, where data is drawn randomly at each step. An example is the Pegasos algorithm for SVMs [80], which at each iteration operates on a single training example. Note that our setting is *offline*; however, if we use an online algorithm, we *simulate* it in an offline setting.

Let r be the ranking function for constraints as defined in Step 1 and let \mathcal{A} be any sequential algorithm that receives the constraints in the order defined by r . Note that the j th local problem contains precisely the variables and constraints that \mathcal{A} considers when deciding the value of x_j .

In order to solve the j th local problem, the constraints Y_j are considered sequentially in the order assigned by the ranking r . At each step, LOCO simulates the arrival of a constraint in Y_j , in the order implied by r , and the variables in X_j are updated. We assume the univariate non-negativity constraints do not arrive sequentially and are known initially. In $|Y_j|$ steps, the algorithm produces some solution for all the variables in X_j , which includes the desired x_j component of the solution. At this point, x_j 's value is identical to its value in the solution produced by \mathcal{A} : the construction of Y_j and X_j guarantees that it has been updated precisely as it would have been in the execution of \mathcal{A} , up to the point when h , the highest ranked constraint that contains x_j arrived. Clearly, x_j will not be updated at any point afterwards, by the definition of the sequential algorithm. As the value of x_j when solving the j th local problem is identical to its value when executing \mathcal{A} on the entire problem (1) for every j , we get the following lemma. A proof is in Appendix A.

LEMMA 2. Let $[x_1^*, \dots, x_N^*]$ be the solution obtained if the sequential algorithm \mathcal{A} is run on the original problem (1), and let \hat{x}_j be the solution obtained by solving the j th local problem (5). Then if $[x_1^*, \dots, x_N^*]$ is an $h(n, m)$ -approximate solution to (1) then $[\hat{x}_1, \dots, \hat{x}_N]$ is also an $h(n, m)$ -approximate solution to (1).

Contrasting LOCO with classical approaches. From the description above, it is clear that LOCO fundamentally differs from dual decomposition and consensus methods. Dual ascent and

Algorithm 1 LOCO (Local Convex Optimization)

Input: Convex Program of form (1), sequential algorithm \mathcal{A} , ranking $r : [m] \rightarrow [0, 1]$, index of agent j

Output: \hat{x}_j

Initialize: Calculate the rank of the constraints for all i such that $a_{ij} \neq 0$. Let h be the index of the highest ranked constraint: $h = \operatorname{argmax}_j \{r(i) | a_{ij} \neq 0\}$

Step 1: Find sets X_j and Y_j associated with x_j .

$X_j = \emptyset; Y_j = \{h\}$

ptr = 1; endptr = 2

while ptr < endptr **do**

$h = Y_j(\text{ptr})$

 ptr++

for all $j' \in [n]$ s.t. $a_{hj'} \neq 0$ **do**

for all $i \in [m]$ s.t. $a_{ij'} \neq 0$ **do**

if $j' \notin X_j$ **then**

$X_j \leftarrow X_j \cup \{j'\}$

if $r(i) < r(h)$ **then**

if $i \notin Y_j$ **then**

$Y_j \leftarrow Y_j \cup \{i\}$

 endptr++

Step 2: Use \mathcal{A} to solve the *local problem* (5) defined on X_j and Y_j . Constraints arrive in the order determined by r .

consensus methods iterate until *global* optimality conditions are met. In order to check for global optimality, methods such as ADMM typically require communication among all nodes in the distributed network at each iteration. LOCO operates in a completely different way; when constructing sets X_j and Y_j , the j th node only interacts with nodes in X_j , and then solves its local problem without requiring further communication beyond that set of nodes. Thus, communication is strictly localized and there are no multiple rounds of communication.

As a result, there is a difference in the form of the theoretical guarantees for LOCO and dual decomposition/consensus algorithms. Convergence rate bounds are the goal when studying dual decomposition and consensus methods. In contrast LOCO is a framework that inherits the convergence or stopping criterion of the local sequential algorithm employed. LOCO executes for a predetermined number of steps, which is the size of set Y_j . In contrast, for ADMM the number of iterations required is unknown a priori (though it can be bounded). LOCO produces an $h(n, m)$ -approximation to the solution in exactly $|Y_j|$ steps.

Related literature. Distributed optimization is a field with a long history. In the 1960s, approaches emerged for solving large scale convex programs in a distributed manner. Early approaches include [8, 10, 23, 37, 74, 87].

Distributed optimization algorithms can be broadly categorized into dual decomposition methods [86], subgradient methods [59, 62], and proximal gradient methods [81]. Many of these distributed algorithms use consensus methods as a way to distribute computation among the agents. For example, ADMM is a popular dual decomposition method, introduced by Gabay and Mercier [29] that can be implemented in a consensus setting [13]. Variants of consensus ADMM have been studied in the context of support vector machines [28] and generally in distributed model fitting

[26, 31, 58]. ADMM has also found broad applications in denoising images [84] and signal processing [20, 79]. Despite the success of ADMM and other techniques for distributed optimization, they tend to require significant memory storage at each node, and suffer from large communication costs. For example, distributed dual decomposition methods typically requires several rounds of communication between neighbors, and use as many as $O(n)$ messages at each round, where n is the number of nodes in the graph. In our work, we propose a technique that is lighter in both communication and computation, and is more robust to stragglers and the entry/exit of agents.

Within the networked control and communication networks literature, there is a large body of work on distributed algorithms [19, 40, 50]. Dual decomposition algorithms are particularly prominent in this setting. For example, Wei et al. [88] propose a novel approach for solving the network utility maximization problem. See [66] for a survey of distributed algorithms for NUM. Additional recent distributed dual decomposition algorithms include [17, 57].

More broadly, distributed computation is an active field today. Some recent work that is connected to the current paper includes [53], which proposes a distributed decomposition method based on passing gradient information between nodes with the goal of limiting communication. Khirirat et al. [41] also propose a gradient based approach, one in which gradient compression techniques are utilized to improve iteration and communication complexity for the gradient descent algorithm. More recent consensus based asynchronous distributed approaches include [7, 16, 91]. Additionally, Hu et al. [36] introduce a decomposition method which seeks to decrease required communication by solving smaller subproblems at each node. The subproblems are defined on a subset of the variables, which is similar to our approach.

We emphasize that the above approaches and other decomposition based methods [7, 16, 36, 41, 52, 53, 91] differ from our approach in the form of messages passed. The methods discussed above send local copies of the solution vector $x \in \mathbb{R}^n$, or gradient information, to maintain consensus. Thus, these messages are typically a vector in \mathbb{R}^n . We however send small pieces of the constraint matrix, A . Since the A matrix is very sparse, this amounts to only sending several matrix coefficients a_{ij} at a time, along with their index information (i, j) . Thus, our messages are extremely lightweight, and throughout the paper, any comparison via the number of messages to other algorithms is a conservative estimate of the benefits of LOCO.

Another key difference between the approaches used in [36, 41] and LOCO is a trade-off between the cost of sending messages versus the cost of doing heavy computation at each node in the graph. In [36], messages are sent between all neighboring nodes at each iteration, making it relatively message heavy; in contrast, LOCO sends very few messages. In terms of computation, however, [36] does very light computation in each step at each node while LOCO does more computation locally. The choice of which approach to use depends on which is more expensive: communication or computation.

Another way in which our framework differs from dual decomposition is that it does not require every node to converge to the full, global solution, i.e., consensus. In particular, our framework provably produces an α -approximation to the optimal local action in a logarithmic number of steps; whereas dual decomposition and consensus algorithms require analysis of convergence rates and stopping criteria.

Stragglers and failures have been major obstacles for the distributed optimization literature during the past decades. Two prominent goals are (i) the design of asynchronous algorithms and (ii) providing Byzantine faulty tolerance. For both of these goals, the challenge is to be robust to communication delays or unreliability in the system. In asynchronous computation, the goal is to compute the solution when distributed agents do not report updates in a reliable way [63, 67, 87]. In the Byzantine faulty tolerance, some components of the distributed system are unreliable and perhaps adversarial [18]. Our work shares these goals, but approaches them in a different way; we

design a new way to distribute the computation between the agents, requiring less communication and thus approaching robustness to failures differently.

Some of the key insights behind LOCO are based on a field of theoretical computer science: *local computation algorithms* (LCAs) [75]. Most of the focus of research in this field has been on graph problems such as matching, maximal independent set, and coloring [1, 3, 27, 46, 73]. Our work contributes to the LCA literature by moving from graph problems to the more general domain of distributed convex optimization, which has not been studied previously.

Two other related lines of work are the distributed *LOCAL* and *CONGEST* models [69], in which the complexity of a protocol is measured by the number of *rounds* required. Of particular relevance is [43], which concerns solving packing linear programs in a distributed manner in the *LOCAL* model. We note that our algorithm can be implemented in the *LOCAL* (and *CONGEST*) models, in $O(\log n)$ rounds; the algorithm of [43], while using a polylogarithmic number of rounds, can use as much as linear communication if the diameter of the network is small.

Lastly, our approach shares characteristics with *sketching* and *leverage score sampling* [89], in that a subset of the rows of the data matrix A are selected and a problem of smaller dimension is solved. However, our work differs from these approaches significantly. For example, we select a block of the matrix, or a subset of both rows and columns. Thus the dimension of the problem is reduced in both the number of variables and the number of data points.

4 MAIN RESULTS

In this section we provide results that bound the communication demands of LOCO and the quality of the solution it produces. The key insight in the design of LOCO is that it is possible to convert any local sequential algorithm into a distributed algorithm. We prove that the resulting distributed algorithm has the same approximation ratio as the original local sequential algorithm. In particular, our main theoretical result shows that LOCO provides solutions to convex optimization problems that are as close to optimal as those of the best local sequential algorithms for the problems, while using exponentially less communication than classical distributed optimization algorithms. Further, because each agent computes its local piece of the solution without global communication, LOCO provides significant improvements in robustness compared to traditional consensus-based and dual descent-based approaches.

Our main result is summarized in the following theorem.

THEOREM 3. *Let P be a convex problem of form (1), where $A \in \mathbb{R}^{m \times n}$ has sparsity d . Consider LOCO instantiated with a local sequential algorithm \mathcal{A} for P with approximation ratio $h(n, m)$. Each agent $j \in V$, where $|V| = N$, independently computes \hat{x}_j using at most $2^{O(d^2)} q_j \log m$ messages with probability $1 - 1/\text{poly}(m)$. The resulting complete solution $[\hat{x}_1, \dots, \hat{x}_N] \in \mathbb{R}^n$ provides an $h(n, m)$ -approximate solution to P .*

This result shows that there is no performance loss when converting the local sequential algorithm to a distributed algorithm using LOCO. Further, for sparse graphs (where d is a constant), the communication demands are logarithmic, as opposed to linear like in consensus based algorithms.

Theorem 3 provides a general result, but it is also useful to illustrate this result for specific local sequential algorithms. In particular, LOCO can be used broadly for any class of optimization problems for which local sequential algorithms exist. Thus, improvements to local sequential and online algorithms immediately yield improved distributed algorithms.

We illustrate this with the following corollaries for the cases of linear programs and linear SVMs. These two corollaries provide the basis for the case studies for NUM and SVMs in Section 5.

In the case of NUM, we focus on the goal of throughput maximization, which means that the objective is linear. In this case, we can use the online algorithm from [14] for packing linear programs, which yields the following corollary.

COROLLARY 4. *Given a linear program with n variables, m constraints, and a sparse constraint matrix, each agent $j \in V$, where $|V| = N$, independently computes \hat{x}_j using at most $O(\log m)$ messages with probability $1 - 1/\text{poly}(m)$. The resulting complete solution $[\hat{x}_1, \dots, \hat{x}_N]$ provides an $O(\log m)$ -approximation.*

In the case of SVM, we focus on the linear SVM problem described in (4). In that case, we can apply the Pegasos [80] algorithm, which yields the following result.

COROLLARY 5. *Given a linear SVM problem with n variables, m constraints, where d is a bound on the number of nonzero features in each example, and λ is the regularization parameter, each agent $j \in V$, where $|V| = N$, independently computes \hat{x}_j using at most $O(\log m)$ messages with probability $1 - 1/\text{poly}(m)$. The resulting complete solution $[\hat{x}_1, \dots, \hat{x}_N]$ provides an ϵ -approximation.*

Note that we focus on NUM with a linear objective, but LOCO is not limited to linear objectives and Theorem 3 can be applied to NUM with a general convex objective function, for example, using the algorithm in [6].

Now that we have concretely stated both the algorithm and results, we see how LOCO lends itself to the robustness properties outlined in the introduction. As stated in Theorem 3, setting up each local problem requires at most $2^{O(d^2)} \log m$ messages. This bound on communication implies that an agent will only communicate with a logarithmically bounded number of agents, constituting a small neighborhood around the agent. This behavior makes the computation robust to failures or delays – a failure will only effect nearby agents, leaving agents outside of the logarithmically bounded neighborhood unaffected. Similarly, if a new agent enters the system, only agents in the logarithmically bounded neighborhood must share new data and recompute. This is in contrast to the large body of distributed optimization algorithms, which typically require all of the agents to update computations if a new agent enters the system. Additionally, after the initial round of communication, each local problem is solved independently at the corresponding node with no further communication. This increases the robustness of LOCO to failures – the computation is done completely locally, never disrupted by failures or delays.

A final note about these results is that our analysis is based on worst-case *adversarial* input for local sequential algorithms. Thus, it is natural to expect LOCO to achieve a much better approximation ratio in practice, as LOCO randomizes constraint arrival order and so adversarial inputs are extremely unlikely. We verify this intuition in Section 5, confirming that our empirical results outperform the theoretical guarantees by a considerable margin. An interesting open problem is to give better theoretical bounds for the local sequential for stochastic inputs. If such results are obtained they would immediately improve the bounds in Theorem 3.

Proofs. In the remainder of this section we prove the above results. To begin, in order to bound the communication complexity in Theorem 3, the core argument needed is a bound on the size of sets Y_k . First, we need to define some terminology for hypergraphs. Given a hypergraph $\mathcal{H} = (V, H)$, the *neighbors of a hyperedge* $y \in H$, denoted $\mathcal{N}(y)$, are the hyperedges with vertices in common with y . The *hyperedge degree* of y is its number of neighbors, $|\mathcal{N}(y)|$.

Using this terminology, we proceed to prove some technical lemmas.

LEMMA 6. *Let $\mathcal{H} = (V, H)$ be a hypergraph, $|H| = m$, whose hyperedge degree is bounded by d' , and let $r : H \rightarrow [0, 1]$ be a function that assigns to each hyperedge $y \in H$ a number between 0 and 1 independently and uniformly at random. Let Y_{\max} be the size of the largest set of constraints Y_y*

chosen for a local problem: $Y_{max} = \max\{|Y_y| : y \in H\}$. Then, for $\lambda = 4(d' + 1)$,

$$\Pr[|Y_{max}| > 2^\lambda \cdot 15\lambda \log m] \leq \frac{1}{m^2}.$$

The proof of Lemma 6 uses ideas from a proof in [73], and employs a *quantization* of the rank function. Due to space constraints its proof is found in Appendix B.

We are now ready to prove Theorem 3.

PROOF OF THEOREM 3. First, consider the communication complexity. The result in Lemma 6 refers to communication on the hypergraph, \mathcal{H} . However, messages will be sent on the physical network, G . Thus we can set $d' = d^2$ in Lemma 6 to describe communication on the physical network.

Lemma 6 establishes the communication required for an individual agent when computing one scalar component of the solution. However, recall that each agent computes the solution to the vector $x_j \in \mathbb{R}^{q_j}$. Taking the union bound over the size of this vector, we see that $\Pr[|Y_k| > 2^{O(d^2)} q_j \log m] \leq \frac{1}{m^2}$.

Due to the sparsity of the constraint matrix, it holds that $|X_k| < d|Y_k|$. Thus, the number of messages is upper bounded by $|X_k|$, and thus, $\Pr[|X_k| > 2^{O(d^2)} q_j \log m] \leq \frac{1}{m^2}$. Finally, the approximation ratio is established by Lemma 2, completing the proof. \square

In addition to Lemmas 2 and 6, the following technical lemma is needed to complete the proof of Corollary 4. We restate Theorem 14.1 from [14].

LEMMA 7. *For any $B > 0$, there exists a B -competitive online algorithm for linear programs with m constraints; each constraint is violated by a factor at most $\frac{2 \log(1+m)}{B}$.*

PROOF OF COROLLARY 4. The approximation ratio is due to the online algorithm presented and analyzed in [14] (see Lemma 7). Theorem 3 and Lemma 7, setting $B = 2 \log(1 + m)$ imply Corollary 4. \square

PROOF OF COROLLARY 5. The approximation ratio is due to the online algorithm presented and analyzed in [80]. Theorem 3 implies Corollary 5. \square

5 CASE STUDIES

The previous section provides worst-case bounds on the performance of LOCO. Here, we illustrate the performance that can be expected in real applications. To do this, we use both synthetic and real data to look at linear programs, NUM, and SVM. The results demonstrate an orders-of-magnitude reduction in communication with LOCO as compared to ADMM, while maintaining nearly optimal solutions. We demonstrate the performance of our algorithm on linear programs, a network utility maximization problem, an on training support vector machines. Experiments were run on a server with Intel E5-2623V3@3.0GHz 8 cores and 64GB RAM.

5.1 Experimental Setup

5.1.1 Linear Programming. Our first set of experimental results use synthetic linear programming examples. We generate random synthetic instances of linear programs as follows. To generate $A \in \mathbb{R}^{m \times n}$, we set $a_{ij} \sim_{i.i.d.} U(0, 1)$ with probability p and $a_{ij} = 0$ otherwise. We then add $\min\{m, n\}$ i.i.d. draws from $U(0, 1)$ to the main diagonal, to ensure each row of A has at least one nonzero entry. Similarly we set $b_i \sim_{i.i.d.} U[0, 1]$. We set the minimum and maximum transmission rates to be $\underline{x}_i = 0$ and $\bar{x}_i = 1$. Unless otherwise stated, we set $n = m$ and $f_j(x_j) = c_j x_j$ with $c_j \sim_{i.i.d.} U[0, 1]$.

For the case of linear programs in Step 2 of our algorithm, we employ an online algorithm for covering and packing linear programs proposed by [14], pseudocode for which is in Appendix D. Running this algorithm requires tuning one parameter: B , discussed in Lemma 7 in Appendix 4, which governs the worst-case guarantee for the online algorithm used in Step 2. A smaller B gives a better guarantee in terms of message complexity, however some constraints may be violated. Setting $B = 2 \ln(1 + m)$ provides the best worst-case guarantee, and is our choice in the experiments unless stated otherwise. In fact, it is possible to tune B (akin to tuning ADMM) to specific data, as the constraints are often still satisfied for smaller B . In Figure 4 (c), we show the improvement in performance guarantee by tuning B , while keeping the dual solution feasible.

Throughout all experiments, each point in the figures is averaged over 50 executions, and the ranking function r is a random permutation of the vertex IDs.¹

5.1.2 Network Utility Maximization (NUM). Our second set of experiments focus on the linear network utility maximization (NUM) problem. We consider the graph of Autonomous System (AS) relationships in [15]. The graph has 8020 nodes and 36406 edges. To interpret the graph in a NUM framework, associate each source node with a path of edges, ending at a destination node. For each source i in the graph, we randomly select a destination which is at distance ℓ_i , sampled *i.i.d.* from $\text{Unif}[\ell - 0.5\ell, \ell + 0.5\ell]$. Here $f_j(x_j) = c_j x_j$, which corresponds to throughput maximization. We draw $c \in \mathbb{R}^n$ *i.i.d.* from $\text{Unif}[0, 1]$, and set the minimum and maximum transmission rates to be 0 and 1.

Note that Step 2 of LOCO is implemented using the same online algorithm as for linear programming, described above.

5.1.3 Support Vector Machines (SVMs). Our final example is the linear SVM problem, as described in (4). We run experiments on both randomly generated synthetic data, and real data.

For the synthetic data, we define a matrix $Z \in \mathbb{R}^{m \times n}$ as follows. We set $z_{ij} \sim \text{i.i.d. } N(0, 1)$ with probability p and $z_{ij} = 0$ otherwise. We then add $\min\{m, n\}$ *i.i.d.* draws from $N(0, 1)$ to the main diagonal, to ensure each row of Z has at least one nonzero entry.² We set $y_i = +1$ with probability 0.5 and $y_i = -1$ otherwise.

We also run LOCO to train SVMs on the Reuters RCV1 Text Categorization Test Data Set [47], for classification tasks CCAT and C11. This data set has sparsity $p = 0.16\%$, with $n = 47, 236$ features, $m = 781, 265$ training examples, and $m_{\text{test}} = 23149$ testing examples.

When implementing Step 2 of our algorithm for the case of SVM, we employ the well known Pegasos [80] algorithm. Note that in Pegasos, at each step a data point is selected uniformly at random. Our setting is also designed to do this, as the ranking function r is also a collection of values drawn from $[m]$ uniformly at random. However in Pegasos [80], the stopping criterion can be varied along with accuracy requirements, while in our case, we run exactly $|Y_k|$ iterations of Pegasos to solve each local problem. Unless specified, we set the regularization parameter to be $\lambda = 0.0001$.

5.2 Benchmark & Performance Metrics

We use ADMM as a benchmark for comparison in this paper given its prominence in applications. For completeness, the pseudocode for ADMM is included in Appendix E. Running ADMM requires tuning four parameters [13]. Unless otherwise specified, we set the relative and absolute tolerances to be $\epsilon^{\text{rel}} = 10^{-4}$ and $\epsilon^{\text{abs}} = 10^{-2}$, the penalty parameter to be $\rho = 1$, and the maximum number of

¹For the purposes of our simulations, such a permutation can be efficiently sampled, and guarantees perfect randomness. For larger n and m , it is possible to use pseudo-randomness with almost no loss in message complexity [73].

²Note that the sparsity of A is not necessarily a constant; however, this can only increase the message complexity.

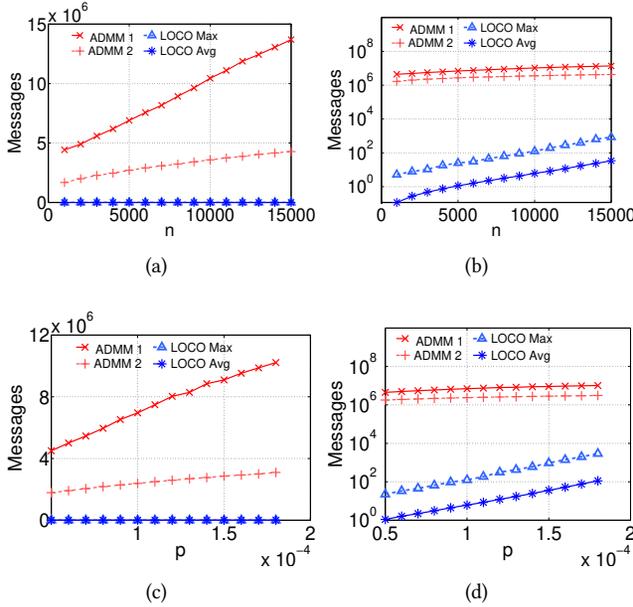


Fig. 3. Messages required by LOCO and ADMM for random linear programming instances. Plots (a) and (b) vary n while fixing sparsity $p = 10^{-4}$, showing the results in linear-scale and log-scale respectively. Plots (c) and (d) fix $n = 10^3$ and vary the sparsity p .

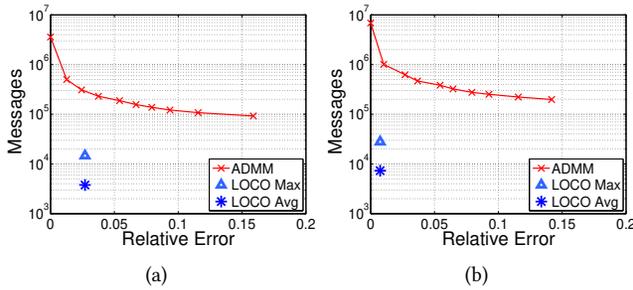


Fig. 4. Comparison of the relative error and messages required by LOCO and ADMM for random linear programming instances. Plots (a) and (b) show the Pareto optimal curve for ADMM for two different settings of the relative tolerance parameter: $\epsilon^{rel} = 10^{-4}$ and $\epsilon^{rel} = 10^{-1}$ respectively.

allowed iterations to be $t_{max} = 10000$. This is done to provide the best performance for ADMM: the parameters are tuned in the typical fashion to optimize ADMM [13].

We evaluate ADMM and LOCO with respect to the quality of the solution provided and the number of messages sent. To assess the quality of the solution we measure the *relative error*, which is defined as $\frac{|p^* - p^{LOCO}|}{|p^*|}$, where p^* is the optimal solution. For problem instances of small dimension, one can run an interior point method to check the optimal solution, but this is tedious for large problem sizes. In the large dimension cases we consider, we regard p^* to be ADMM's solution with small tolerances, such that the maximum number of allowed iterations is never needed. Note

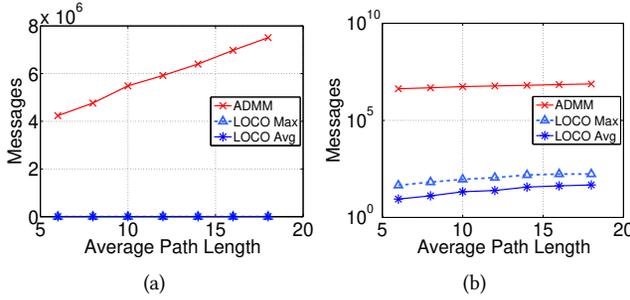


Fig. 5. Illustration of the number of messages required by LOCO and ADMM for NUM using an Autonomous System (AS) graph.

that the relative error is an empirical, normalized version of the approximation ratio for a given instance.

We now explain how we count the number of *messages* used by each of the algorithms. As defined in Section 2, a message is a list of matrix coefficients $\{a_{ij} \forall j \in [n] : a_{ij} \neq 0\}$, for a given i th row of the matrix A , along with the coefficient b_i . Since the A matrix is very sparse, this amounts to only sending several matrix coefficients a_{ij} at a time. In contrast, a message in ADMM passes a local copy of the primal and dual solution vectors, which are vectors in \mathbb{R}^n and \mathbb{R}^m . Thus, the size of the messages passed by LOCO is smaller than that of ADMM. Our comparisons based on the number of messages is a conservative estimate for the improved communication efficiency of LOCO.

For a distributed implementation of ADMM, two sets of n variables are updated on separate processors (see Chapter 7.1 of [13]). The number of messages required by ADMM is twice the number of nodes in the network G , multiplied by the number of iterations required by ADMM. In contrast, LOCO communicates only in order to construct the local problems; running the online algorithm does not require any communication. The number of messages required to construct the k th local problem is proportional to the size of set X_k . When communicating over the hypergraph \mathcal{H} , at most $|X_k| + d$ messages are required, and over any general network G , at most $d^2(|X_k| + d)$ messages are required.

Finally, we compare the running times of LOCO and ADMM. We define the *speedup* as the running time of ADMM divided by the running time of LOCO. In all cases, we allow the n nodes to compute in parallel.

5.3 Experimental Results

This section describes the results for our case studies. In each case our results highlight order-of-magnitude reductions in communication overhead compared to ADMM with minimal decrease in accuracy. Further, this happens while providing significantly improved robustness.

5.3.1 Linear Programming. Our first experimental results focus on synthetic examples of linear programs. Figure 3 illustrates our results, showing that LOCO requires considerably fewer messages than ADMM, across both small and large n and varying levels of sparsity. In these plots, we also chose to plot not only the average messages over all the subproblems, LOCOAvg, but also the maximum amount, LOCOMax, for the problem with the largest sets X_k and Y_k .

The performance of ADMM depends significantly on the tolerance used, and so the figure includes ADMM with tolerances ϵ^{rel} of both 10^{-4} (ADMM 1) and 10^{-3} (ADMM 2). Note that even with

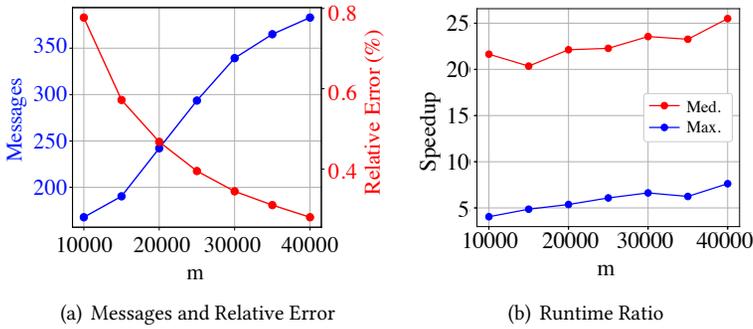


Fig. 6. Plot (a) illustrates of the number of messages required by LOCO and the relative error between LOCO and ADMM, in the case of synthetic SVM data, when $n = 10,000$ and m is varied, and $p = 0.03\%$. In all instances, the number of messages required by ADMM was over 100K, an order of magnitude larger than LOCO, and are not plotted due to being out of range for the plot. Plot (b) illustrates the speedup provided by LOCO compared to ADMM. ‘Max’ and ‘Med’ refer to the largest and median sized subproblem respectively.

suboptimal tolerance, which results in fewer iterations, ADMM still requires orders of magnitude more communication than LOCO.

We additionally explore the tradeoff between message complexity and relative error. Figures 4(a) and (b) illustrate the Pareto optimal frontier for ADMM: the minimal messages needed in order to obtain a particular relative error. We tune the parameters of ADMM such that the algorithms have comparable relative error to enable a fair comparison. Unlike ADMM, LOCO does not have a comparable parameter to tune, thus LOCO corresponds to a single point in the figures. This point is beyond the Pareto frontier of ADMM, highlighting the order-of-magnitude reduction in communication provided by LOCO. In all the plots, we note that the standard deviations are small enough that they are not visible on the plots.

In all of these plots, remember that ADMM is doing “more” than LOCO. These plots show the communication necessary for an agent to compute its local action. However, under ADMM the agent is computing the full, global solution, while in LOCO the agent is computing precisely what is desired – the local action of the agent.

5.3.2 Network Utility Maximization (NUM). Our second set of results focuses on throughput maximization in NUM. Figure 5 demonstrates and order of magnitude difference in the messages required by LOCO compared to ADMM. The number of messages is shown as a function of the average path length in the instances of the NUM problems. Here, the average path length serves as a metric to describe the sparsity of the constraint matrix, as it has a nonzero component for every utilized edge in the graph. LOCO greatly outperforms ADMM for all tested average path lengths. In all instances, the relative error was 0.4% or less, and so the improvement comes with minimal cost in terms of accuracy. Similar results hold for other objectives beyond throughput maximization, but we omit these due to space constraints.

5.3.3 Support Vector Machines (SVMs). To evaluate LOCO in the context of SVMs, we use both synthetic and real data. Our first set of experiments focus on synthetic data and illustrate the number of messages required and the quality of the solution produced by LOCO compared to ADMM. Figure 6(a) shows both the number of messages for LOCO and relative error between the LOCO and ADMM as m varies. The messages are averaged over the local problems. The messages required by for ADMM are all above 100K, and out of range for this plot. In general ADMM requires

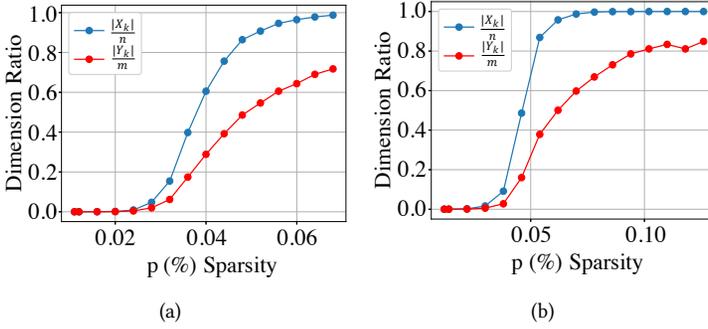


Fig. 7. Comparison of the local problem dimension ($|X_k| \times |Y_k|$) to the original problem size ($m \times n$), averaged over all the local problems, in the case of synthetic SVM data. In (a) $n = m = 10,000$, in (b) $n = 5,000$ and $m = 10,000$.

Table 1. SVM on CCAT and C11 from Reuters RCV1

Alg.	CCAT Train (Test)	C11 Train (Test)	Mess- ages
ADMM	0.31% (10.74%)	0.14% (3.09%)	330K
LOCO	4.84% (7.88%)	2.64% (3.01%)	18K

an order of $O(nT)$ messages, where T is the number of iterations. We also see that as the problem sizes increases, the relative error between the LOCO and ADMM decreases. Figure 6(b) shows the speedup provided by LOCO compared to ADMM. As the problem size m increases, the speedup increases.

Next, we evaluate the performance of LOCO on real data using the Reuters RCV1 Text Categorization Test Data Set [47]. Results are found in Table 1.

We found that for task CCAT LOCO produces a test error of 6.16% when run on the original dataset. However, when generating sets Y_k , we found that the dataset could be thresholded to increase sparsity and reduce communication overhead further. To do this, we thresholded the values in the matrix below 0.1, setting all such values equal to 0. The thresholding value was chosen so that the test error did not change significantly, while the sparsity of the resulting matrix decreased. We tried several different thresholds, and found 0.1 to be representative for this dataset. We note that thresholding is a valuable tool in practice only when it does not increase the test error significantly. This resulted in a matrix with sparsity $p = 0.045\%$. Running LOCO on this thresholded data set led to a slight increase in the test error; 6.16% originally, and 7.88% with thresholding. However, the local problems reduced to an average size of $|Y_k| = 18K$, which is a order of magnitude reduction of the original problem dimension of $m = 781,265$, and consequently yields an order-of-magnitude reduction in communication.

5.3.4 Sparsity. The performance of LOCO is dependent on the sparsity of the constraint matrix A . As seen in Figure 3 (c) and (d), the number of messages increases as p increases. Many real world problems, such as NUM and SVM discussed above, involve very sparse matrices which are appropriate for LOCO. We further investigate the effects of sparsity in Figure 7. The figure highlights that the improvement in communication achieved by LOCO is possible because the

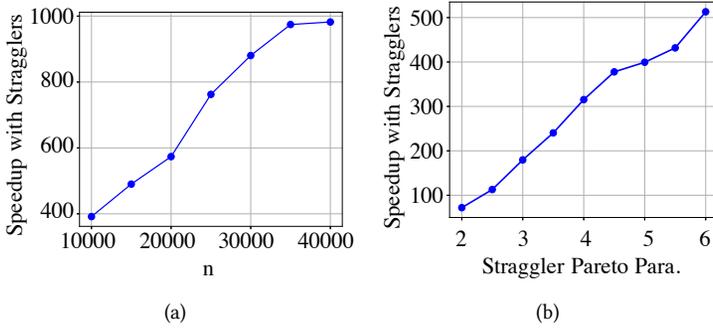


Fig. 8. Illustration of the impact of stragglers. The plots illustrate how the speedup of LOCO relative to ADMM varies with (a) the problem dimension n , when the Pareto shape parameter is set to 5, and (b) the shape parameter of the straggler distribution. Synthetic SVM data is used with $n = m = 10,000$ and $p = 0.03\%$.

dimensions of the local problems, $(|X_k| \times |Y_k|)$, are small compared to the original $(m \times n)$ problem. The dimensions of the local problems are dependent on the sparsity, p , due to the way in which we use the sparsity structure to determine when new constraints are added to set Y_k in Algorithm 1. In Figure 7, as p increases, the local problems get larger, and plateau when $|Y_k| = m$. For the data described in Section 5.1.3, we empirically observe a phase transition like behavior as p varies. We note that this transition depends on the distribution of the placement of the nonzero elements in matrix A . For example in Figure 7(a) and (b), $|Y_k|$ varies differently for different n and m .

5.3.5 Stragglers & Failures. Our last results highlight the robustness of LOCO to stragglers and failures. In modern distributed systems, stragglers are a fact of life. Conflicts and congestion lead to unpredictable delays in local parts of the system, which can then delay the progression of distributed algorithms globally. Figure 8 illustrates the robustness of LOCO to stragglers by plotting the speedup of LOCO as compared to ADMM. In these experiments, we model the distribution of delays caused by stragglers using a Pareto distribution, which is motivated by empirical studies of stragglers in real systems such as [4]. The figure highlights that, as n increases, the speedup provided by LOCO is more pronounced and that as the tail of the distribution of stragglers becomes heavier the difference becomes less pronounced.

We also consider the effect of node failures on LOCO. In LOCO a failure at node j affects all the nodes that share common nodes found in set X_j . We experimented with a variety of settings, and the comparison between ADMM and LOCO is dramatic. A representative example is with $n = m = 10,000$ and $p = 0.03$. The results from the other settings we considered are qualitatively the same. In this setting, the largest set X_j in LOCO has about 5% of all the nodes. As a result, the failure of a single node has the capacity, in the worst case, to affect about 15% of the nodes. In contrast, in ADMM, a single failure stops the whole process as the central node waits for the failed node, and thus no nodes obtain solutions.

6 CONCLUDING REMARKS

We introduced a new approach for the design of multi-agent systems using distributed optimization based on ideas from the emerging field of local computation algorithms. In our framework, LOCO, each agent in a network computes its local piece of the solution, using exponentially less communication than existing techniques, and produces a provably nearly optimal solution without the need for iterative rounds of communication. Additionally, LOCO is robust to network stragglers and

failures due to the independent nature of the local problems. Our empirical case studies demonstrate that LOCO requires orders of magnitude fewer messages than ADMM, while maintaining high quality solutions in random linear programming instances, and NUM and SVM problems.

We remark that the reduction in the paper holds for worst-case guarantees of sequential algorithms, when the constraints arrive in adversarial order. However, in LOCO we determine the order of arrival internally, and so the worst-case guarantees may be too conservative. Our reduction also holds for average-case guarantees of sequential algorithms, when the constraints arrive uniformly at random. Hence, any such guarantees immediately apply to LOCO. Currently, there are few such theoretical guarantees to problems to which LOCO is applicable, and we believe this is an interesting research direction. Further, it may be possible to improve performance by optimizing the order in which constraints arrive or by choosing the form of randomness used in the order of arrivals in order to avoid the adversarial behavior underlying the worst-case bounds in this paper.

We view this paper as a first step towards the investigation of local computation algorithms for distributed optimization. In future work, we plan to study the performance of LOCO on more general network optimization problems. Further, it would be interesting to apply other techniques from the field of local computation algorithms to develop algorithms for other settings in which distributed computing is useful, such as power systems and federated machine learning.

ACKNOWLEDGMENTS

This work was funded by the National Research Foundation through the AitF-1637598, CNS-1518941, and CNS-1254169 grants, along with the Linde Foundation and an Amazon AWS Artificial Intelligence Fellowship.

A PROOF OF LEMMA 2

By the definition of local sequential algorithms, the last time a variable x_j can be updated is the last arrival time of a constraint y_i such that $a_{ij} \neq 0$ and its new values depend *only* on the value (at step i) of the primal variables x_j such that $a_{ij} \neq 0$. Assume that LOCO simulates \mathcal{A} , a local sequential algorithm. It suffices to show that when y_i “arrives” during the execution of LOCO, the primal variables x_j such that $a_{ij} \neq 0$ have the same value as they do when y_i arrives during the execution of \mathcal{A} . We show this by contradiction.

Denote the constraints in Y_j by $y_1, \dots, y_{|Y_j|}$, and assume that they are sorted by arrival time, i.e., y_1 arrives first out of the constraints in Y_j . Let i' be the smallest value such that there exists some x_j for which $a_{i'j} \neq 0$ that has a different value in the two executions when $y_{i'}$ arrives. If x_j was never updated in the execution, this is because there exists no constraint $y_{i''}$ that is a neighbor of $y_{i'}$ in H that arrived before $y_{i'}$, hence x_j was never updated in the execution of \mathcal{A} it was not updated by \mathcal{A} before $y_{i'}$ arrived. Otherwise, consider the last time x_j was updated by LOCO. Assume this was when $y_{i''}$ arrived. As i' is the smallest value such that there exists some x_j for which $a_{i'j} \neq 0$ that has a different value in the two executions when $y_{i'}$ arrives, it must hold that all of the variables x_j such that $a_{i''j} \neq 0$ were correctly valued, but then by the definition of the local sequential algorithm, $x_{j'}$ must have been updated correctly, a contradiction.

B PROOF OF LEMMA 6

Logarithms are base e . Let $\mathcal{H} = (V, H)$ be a hypergraph. Recall that the *neighbors of a hyperedge* $y \in H$ are the hyperedges with vertices in common with y , denoted $\mathcal{N}(y)$. For any set of hyperedges $S \subseteq H$, let $\mathcal{N}(S)$ denote the set of hyperedges that are not in S but are neighbors of some hyperedge in S : $\mathcal{N}(S) = \{\mathcal{N}(y) : y \in S\} \setminus S$. For a set $S \subseteq H$ and a function $g : H \rightarrow \mathbb{N}$, we use $S \cap g^{-1}(i)$ to denote the set $\{y \in S : g(y) = i\}$.

Let $\mathcal{H} = (V, H)$ be a hypergraph, and let $g : H \rightarrow \mathbb{N}$ be some function on the hyperedges. An *adaptive hyperedge exposure procedure* is one that does not know g a priori. The procedure is given an edge $y \in H$ and $g(y)$. Edges from $H \setminus S$ are iteratively added to S ; for every edge y' added, $g(y')$ is revealed immediately after y' is added. Let S_t denote S after the addition of the t -th edge. The following is a concentration bound that shows that for a random g , any sufficiently large set of adaptively exposed hyperedges, less than half will have the same value of g w.h.p. Its short proof is given for completeness.

LEMMA 8. *Let $\mathcal{H} = (V, H)$ be a hypergraph for which $|H| = m$, let $Q > 0$ be some constant, let $\gamma = 15Q$, and let $g : H \rightarrow [Q]$ be a function chosen uniformly at random from all such possible functions. Consider an adaptive hyperedge exposure procedure that is initialized with an edge $y \in H$. Then, for any $q \in [Q]$, the probability that there is some t , $\gamma \log m \leq t \leq m$ for which $|S_t \cap g^{-1}(q)| > \frac{2|S_t|}{Q}$ is at most $\frac{1}{m^4}$.*

PROOF. Let y_i be the i th edge added to S by the adaptive hyperedge exposure procedure, and let I_i be the indicator variable whose value is 1 iff $g(y_i) = q$. For any $t \leq m$, $\mathbb{E} \left[\sum_{i=1}^t I_j \right] = \frac{t}{Q}$. As I_i and I_j are independent for all $i \neq j$, by the Chernoff bound, for $\gamma \log m \leq t \leq m$,

$$\Pr \left[\sum_{i=1}^t I_j > \frac{2t}{Q} \right] \leq e^{-\frac{t}{3Q}} \leq e^{-5 \log m}.$$

A union bound over all possible values of $t : \gamma \log m \leq t \leq m$ completes the proof. \square

Recall that d' is the upper bound on the hyperedge degree. Let $r : V \rightarrow [0, 1]$ be a function chosen uniformly at random from all such possible functions. Partition $[0, 1]$ into $Q = 4(d' + 1)$ segments of equal measure, W_1, \dots, W_Q . For every $v \in V$, set $g(v) = q$ if $r(v) \in W_q$ (g is a quantization of r).

Consider the following method of generating two sets of vertices: Y and Z , where $Y \subseteq Z$. Set Z can be thought of as a set S_t for some t as described in Lemma 8. For some edge h , set $Y = Z = \{h\}$. Continue inductively: choose some edge $w \in Y$, add all $\mathcal{N}(w)$ to Z and compute $g(u)$ for all $u \in \mathcal{N}(w)$. Add the edges u such that $u \in \mathcal{N}(w)$ and $g(u) \geq g(w)$ to Y . The process ends when no more edges can be added to Y .

Y is generated with respect to g , the quantization of r . The actual sets of constraints constructed in LOCO for the local problems are defined with respect to r . Here, $|Y|$ is an upper bound on the size of the sets constructed in LOCO. It is difficult to reason about the size of Y directly, as the ranks of its edges are not independent. The edges of the vertices in Z , though, are independent, as Z is generated by an adaptive hyperedge exposure procedure. Z is a superset of Y that includes Y and its boundary, hence $|Z|$ is also an upper bound on the size of the query set.

We now define $Q + 1$ "layers" - $Y_{\leq 0}, \dots, Y_{\leq Q} : Y_{\leq q} = Y \cap \bigcup_{i=0}^q g^{-1}(i)$. That is, $Y_{\leq q}$ is the set of vertices in Y whose rank is at most q . (The range of g is $[Q]$, hence $Y_{\leq 0}$ will be empty, but we include it to simplify the proof.)

CLAIM 9. *Set $Q = 4(d' + 1)$, $\gamma = 15Q$. Assume without loss of generality that $g(v) = 0$. Then for all $0 \leq i \leq Q - 1$,*

$$\Pr[|Y_{\leq i}| \leq 2^i \gamma \log m \wedge |Y_{\leq i+1}| \geq 2^{i+1} \gamma \log m] \leq \frac{1}{m^4}.$$

PROOF. For all $0 \leq i \leq Q$, let $Z_{\leq i} = Y_{\leq i} \cup \mathcal{N}(Y_{\leq i})$. Note that

$$Z_{\leq i} \cap g^{-1}(i) = Y_{\leq i} \cap g^{-1}(i), \tag{6}$$

because if there had been some $u \in \mathcal{N}(Y_{\leq i})$, $g(u) = i$, u would have been added to $Y_{\leq i}$.

Note that $|Y_{\leq i}| \leq 2^i \gamma \log m \wedge |Y_{\leq i+1}| \geq 2^{i+1} \gamma \log m$ implies that

$$|Y_{\leq i+1} \cap g^{-1}(i+1)| > \frac{|Y_{\leq i+1}|}{2}. \quad (7)$$

In other words, the majority of vertices $v \in Y_{\leq i+1}$ must have $g(v) = i+1$.

Given $|Y_{\leq i+1}| > 2^{i+1} \gamma \log m$, it holds that $|Z_{\leq i+1}| > 2^{i+1} \gamma \log m$ because $Y_{\leq i+1} \subseteq Z_{\leq i+1}$. Furthermore, $Z_{\leq i+1}$ was constructed by an adaptive hyperedge exposure procedure and so the conditions of Lemma 8 hold for $Z_{\leq i+1}$. From Equations (6) and (7) we get

$$\begin{aligned} & \Pr[|Y_{\leq i}| \leq 2^i \gamma \log m \wedge |Y_{\leq i+1}| \geq 2^{i+1} \gamma \log m] \\ & \leq \Pr \left[|Z_{\leq i+1} \cap g^{-1}(i+1)| > \frac{|Y_{\leq i+1}|}{2} \right] \\ & \leq \Pr \left[|Z_{\leq i+1} \cap g^{-1}(i+1)| > \frac{2|Z_{\leq i+1}|}{Q} \right] \\ & \leq \frac{1}{m^4}, \end{aligned}$$

where the second inequality is because $|Z_{\leq i+1}| \leq (d+1)|Y_{\leq i+1}|$, as G 's degree is at most d' ; the last inequality is due to Lemma 8. \square

LEMMA 10. Set $Q = 4(d'+1)$. Let $\mathcal{H} = (V, H)$ be a hypergraph with degree bounded by d' , where $|H| = m$. For any edge $h \in H$, $\Pr[Y_h > 2^Q \cdot 15Q \log m] < \frac{1}{m^3}$.

PROOF. To prove Lemma 10, we need to show that, for $\gamma = 15Q$,

$$\Pr[|Y_{\leq Q}| > 2^L \gamma \log m] < \frac{1}{m^3}.$$

We show that for $0 \leq i \leq Q$, $\Pr[|Y_{\leq i}| > 2^i \gamma \log m] < \frac{1}{m^4}$, by induction. For the base of the induction, $|S_0| = 1$, and the claim holds. For the inductive step, assume that $\Pr[|Y_{\leq i}| > 2^i \gamma \log m] < \frac{1}{m^4}$. Then, denoting by \mathbb{I} the event $|Y_{\leq i}| > 2^i \gamma \log m$ and by $\bar{\mathbb{I}}$ the event $|Y_{\leq i}| \leq 2^i \gamma \log m$,

$$\begin{aligned} & \Pr[|Y_{\leq i+1}| > 2^{i+1} \gamma \log m] \\ & = \Pr[|Y_{\leq i+1}| > 2^{i+1} \gamma \log m : \mathbb{I}] \Pr[\mathbb{I}] \\ & \quad + \Pr[|Y_{\leq i+1}| > 2^{i+1} \gamma \log m : \bar{\mathbb{I}}] \Pr[\bar{\mathbb{I}}]. \end{aligned}$$

From the inductive step and Claim 9, using the union bound, the lemma follows. \square

Applying a union bound over all the hyperedges gives that the size of set Y_k pertaining to the k th local problem (5) is $O(\log m)$ with probability at least $1 - 1/m^2$, completing the proof of Lemma 6.

C A NOTE ON RANKING THE CONSTRAINTS

In the Introduction, we describe generating a random permutation over the constraints. However, storing a random permutation requires $\Omega(n)$ space, and we would need to store this permutation on every node. Instead, we can approximate a random permutation with a random ordering by assigning a real number uniformly at random to each constraint, using function r as we described in Section 3. We note that in practice, such an r does not exist. It has been shown in e.g., [3, 73] that r can be approximated arbitrarily well by a hash function by a random hash function of polylogarithmic length. We do not formally define what we mean by ‘‘arbitrarily well’’ here; we refer the reader to [73] for an in depth discussion. In this paper we assume for simplicity that each node has access to an r function.

Algorithm 2 General Online Fractional Linear Packing

Input: Linear Program defined on $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, and approximation parameter B

Output: x, y

Initialize: $x = \mathbf{0}_n, y = \mathbf{0}_m$

for $i = 1 \dots m$ **do**

for $j = 1 \dots n$ **do**

$$a^j(\max) \leftarrow \max_{k=1}^i \{a_{kj}\}$$

while $\sum_{j=1}^n a_{ij}x_j < 1$ **do**

 Increase y_i continuously

for $j = 1 \dots n$ **do**

$$\delta = \exp\left(\frac{B}{2c_i} \sum_{k=1}^i a_{kj}y_k\right) - 1$$

$$x_j = \max\left\{x_j, \frac{1}{na^j(\max)}\delta\right\}$$

D PSEUDOCODE FOR GENERAL ONLINE FRACTIONAL PACKING

In our experiments in Section 5 we use an online algorithm from [14] for the cases of linear programming and NUM. For completeness we give the details of the algorithm in Algorithm 2.

In this online problem, constraints arrive in an online fashion over a sequence of rounds. During the i th round, the packing variable y_i and the covering variables $x_j \forall j \in [n]$ for which $a_{ij} > 0$ are increased. The minimum y_i is found such that the covering constraints are satisfied.

E ADMM

In our numerical results we compare LOCO to ADMM. For completeness, we describe the application of ADMM to problem (1).

To apply ADMM, we introduce a slack variable $s \geq 0$ such that the inequality constraint becomes $Ax - s = b$. Let $x' = [x, s]^T$, $A' = [A \quad -I]$ and $b = [1_n, \mathbf{0}_n]^T$ where this notation indicates a stack of vectors. We can now write the problem in standard ADMM form,

$$\begin{aligned} \min_{x', z} \quad & g(x') + h(z) \\ \text{s.t.} \quad & x' - z = 0 \end{aligned}$$

where $g = (x)_+$ is the indicator function associated with the constraints $x \geq 0$ and $h(z') = -\sum_{j=1}^n f_j(z_j)$ where $\text{dom } h = \{z | A'z = b'\}$.

Writing down the scaled augmented Lagrangian $L_\rho(x', z, u) = g(x') + h(z) + u^T(z - x') + \frac{\rho}{2} \|x' - z\|^2$, we can see that all the update steps have closed form solution (see Chapter 5.2 of [13]). The updates become:

$$\begin{aligned} x'^{k+1} &= (z^{k+1} + u^k)_+ \\ z^{k+1} &= \begin{bmatrix} \rho I & A'^T \\ A' & 0 \end{bmatrix}^{-1} \begin{bmatrix} \rho(x'^k - u^k) - b \\ c' \end{bmatrix} \\ u^{k+1} &= u^k + (x'^{k+1} - z^{k+1}) \end{aligned}$$

The solution to problem (1) is recovered from the first n entries of x' .

REFERENCES

- [1] Dimitris Achlioptas, Themis Gouleakis, and Fotis Iliopoulos. 2018. Local Computation Algorithms for the Lovász Local Lemma. *CoRR* abs/1809.07910 (2018). <http://arxiv.org/abs/1809.07910>
- [2] Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander Smola. 2012. Scalable inference in latent variable models. In *Proc. of the 5th ACM WSDM*. 123–132.
- [3] Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. 2012. Space-Efficient Local Computation Algorithms. In *Proc. 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1132–1139.
- [4] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. 2014. GRASS: Trimming Stragglers in Approximation Analytics. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*.
- [5] Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcraft, Vahab S. Mirrokni, and Shang-Hua Teng. 2008. Local Computation of PageRank Contributions. *Internet Mathematics* 5(1–2) (2008), 23–45.
- [6] Yossi Azar, Niv Buchbinder, T-H. Hubert Chan, Shahar Chen, Ilan Reuven Cohen, Anupam Gupta, Zhiyi Huang, Ning Kang, Viswanath Nagarajan, Joseph (Seffi) Naor, and Debmalya Panigrahi. 2016. Online algorithms for covering and packing problems with convex objectives. In *Proc. of the IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 148–157.
- [7] Amrit Singh Bedi and Ketan Rajawat. 2017. Asynchronous Incremental Stochastic Dual Descent Algorithm for Network Resource Allocation. *IEEE Transactions on Signal Processing* 66 (2017), 2229–2244.
- [8] Jacobus F. Benders. 1962. Partitioning procedures for solving mixed-variables programming problems. *Numer. Math.* 4, 1 (1962), 238–252.
- [9] Dimitri P. Bertsekas. 1999. *Nonlinear programming*. Athena Scientific.
- [10] Dimitri P. Bertsekas and John N. Tsitsiklis. 1989. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall.
- [11] Vincent D. Blondel, Julien M. Hendrickx, Alex Olshevsky, and John N. Tsitsiklis. 2005. Convergence in multiagent coordination, consensus, and flocking. In *Proc. of IEEE Conference on Decision and Control*. 2996–3000.
- [12] Sem Borst, Varun Gupta, and Anwar Walid. 2010. Distributed caching algorithms for content distribution networks. In *Proceedings of IEEE INFOCOM*. 1–9.
- [13] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. and Trends in Machine Learning* 3, 1 (2011), 1–122.
- [14] Niv Buchbinder and Joseph (Seffi) Naor. 2009. The Design of Competitive Online Algorithms via a Primal-Dual Approach. *Foundations and Trends in Theoretical Computer Science* 3, 2-3 (2009), 93–263.
- [15] CAIDA. 2007. The CAIDA UCSD AS Relationship Dataset <http://www.caida.org/data/as-relationships/> (2007).
- [16] Tsung-Hui Chang, Mingyi Hong, Wei-Cheng Liao, and Xiangfeng Wang. 2016. Asynchronous Distributed ADMM for Large-Scale Optimization—Part I: Algorithm and Convergence Analysis. *IEEE Transactions on Signal Processing* 64, 12 (2016), 3118–3130.
- [17] Nikolaos Chatzipanagiotis, Darinka Dentcheva, and Michael M. Zavlanos. 2015. An Augmented Lagrangian Method for Distributed Optimization. *Math. Program.* 152, 1-2 (2015), 405–434.
- [18] Yudong Chen, Lili Su, and Jiaming Xu. 2017. Distributed Statistical Machine Learning in Adversarial Settings: Byzantine Gradient Descent. *Proc. ACM Meas. Anal. Comput. Syst. SIGMETRICS* 1, 2 (2017).
- [19] Mung Chiang, Steven H Low, A Robert Calderbank, and John C Doyle. 2007. Layering as optimization decomposition: A mathematical theory of network architectures. *Proc. IEEE* 95, 1 (2007), 255–312.
- [20] Patrick L. Combettes and Valerie R. Wajs. 2005. Signal recovery by proximal forward-backward splitting. *Multiscale Modeling & Simulation* 4, 4 (2005), 1168–1200.
- [21] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (1995), 273–297.
- [22] George Dantzig. 2016. *Linear programming and extensions*. Princeton university press.
- [23] George B. Dantzig and Philip Wolfe. 1960. Decomposition principle for linear programs. *Oper. Res.* 8, 1 (1960), 101–111.
- [24] Reza Eghbali and Maryam Fazel. 2016. Designing smoothing functions for improved worst-case competitive ratio in online optimization. In *Neural Information Processing Systems*. 3287–3295.
- [25] Tomaso Erseghe. 2014. Distributed optimal power flow using ADMM. *IEEE Trans. on Power Sys.* 29, 5 (2014), 2370–2380.
- [26] Tomaso Erseghe, Davide Zennaro, Emiliano Dall’Anese, and Lorenzo Vangelista. 2011. Fast Consensus by the Alternating Direction Multipliers Method. *IEEE Trans. on Sig. Proces.* 59 (2011), 5523–5537.
- [27] Uriel Feige, Boaz Patt-Shamir, and Shai Vardi. 2018. On the Probe Complexity of Local Computation Algorithms. In *45th International Colloquium on Automata, Languages, and Programming, (ICALP)*. 50:1–50:14.
- [28] Pedro A Forero, Alfonso Cano, and Georgios B. Giannakis. 2010. Consensus-Based Distributed Support Vector Machines. *J. Mach. Learn. Res.* 11 (2010), 1663–1707.

- [29] Daniel Gabay and Bertrand Mercier. 1976. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications* 2, 1 (1976), 17–40.
- [30] Lingwen Gan, Ufuk Topcu, and Steven H. Low. 2013. Optimal decentralized protocol for electric vehicle charging. *IEEE Transactions on Power Systems* 28, 2 (2013), 940–951.
- [31] Tom Goldstein, Gavin Taylor, Kawika Barabın, and Kent Sayre. 2016. Unwrapping ADMM: Efficient Distributed Computing via Transpose Reduction. In *AISTATS*. 1151–1158.
- [32] Joseph E. Gonzalez, Yucheng Low, Carlos E. Guestrin, and David O’Hallaron. 2009. Distributed parallel inference on large factor graphs. In *Proc. of the 25th Conf. on UAI*. 203–212.
- [33] Carlos Guestrin, Peter Bodik, Romain Thibaux, Mark Paskin, and Samuel Madden. 2004. Distributed regression: an efficient framework for modeling sensor network data. In *Proceedings of the 3rd ACM IPSN*. 1–10.
- [34] Yi Guo and Lynne E Parker. 2002. A distributed and optimal motion planning approach for multiple mobile robots. In *Robotics and Automation, 2002. Proceedings. ICRA’02. IEEE International Conference on*, Vol. 3. 2612–2619.
- [35] Tamir Hazan, Amit Man, and Amnon Shashua. 2008. A parallel decomposition solver for SVM: Distributed dual ascend using fenchel duality. In *Proc. of CVPR*. 1–8.
- [36] Jianghai Hu, Yingying Xiao, and Ji Liu. 2018. Distributed Algorithms for Solving Locally Coupled Optimization Problems on Agent Networks. In *Decision and Control (CDC), 2007 IEEE Annual Conference on*. IEEE, 2420–2425.
- [37] Hugh Everett III. 1963. Generalized Lagrange multiplier method for solving problems of optimum allocation of resources. *Operations research* 11, 3 (1963), 399–417.
- [38] Thorsten Joachims. 2006. Training Linear SVMs in Linear Time. In *Proceedings of the 12th ACM SIGKDD*. 217–226.
- [39] Jonathan Katz and Luca Trevisan. 2000. On the efficiency of local decoding procedures for error-correcting codes. In *Proc. 32nd Annual ACM Symposium on the Theory of Computing (STOC)*. 80–86.
- [40] Frank P. Kelly, A. K. Maulloo, and David K. H. Tan. 1998. Rate control for communication networks: shadow prices, proportional fairness and stability. *J. of the Operational Research Society* 49, 3 (1998), 237–252.
- [41] Sarit Khirirat, Mikael Johansson, and Dan Alistarh. 2018. Gradient compression for communication-limited convex optimization. In *2018 IEEE Conference on Decision and Control (CDC)*. 166–171.
- [42] Jakub Konecny, H. Brendan McMahan, and Daniel Ramage. 2015. Federated Optimization: Distributed Optimization Beyond the Datacenter. *ArXiv abs/1511.03575* (2015).
- [43] Christos Koufogiannakis and Neal E. Young. 2011. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing* 24, 1 (2011), 45–63.
- [44] Yoshiaki Kuwata and Jonathan P How. 2011. Cooperative distributed robust trajectory optimization using receding horizon MILP. *IEEE Transactions on Control Systems Technology* 19, 2 (2011), 423–431.
- [45] Leon S Lasdon. 1970. *Optimization theory for large systems*. Courier Corporation.
- [46] Reut Levi and Ronitt Rubinfeld and Anak Yodpinyanee. 2015. Brief Announcement: Local Computation Algorithms for Graphs of Non-Constant Degrees. In *Proc. of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, (SPAA)*. 59–61.
- [47] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. 2004. RCV1: A New Benchmark Collection for Text Categorization Research. *Journal of Machine Learning Research*. 5 (2004), 361–397.
- [48] Na Li, Lijun Chen, and Steven H Low. 2011. Optimal demand response based on utility maximization in power networks. In *IEEE Power and Energy Society General Meeting*. 1–8.
- [49] Ying Liao, Huan Qi, and Weiqun Li. 2013. Load-balanced clustering algorithm with distributed self-organization for wireless sensor networks. *IEEE Sensors Journal* 13, 5 (2013), 1498–1506.
- [50] Steven H. Low and David E. Lapsley. 1999. Optimization flow control. I. Basic algorithm and convergence. *IEEE/ACM Transactions on Networking* 7, 6 (1999), 861–874.
- [51] Steven H Low, Fernando Paganini, and John C Doyle. 2002. Internet congestion control. *IEEE Control Systems* 22, 1 (2002), 28–43.
- [52] Sindri Magnússon, Chinwendu Enyioha, Na Li, Carlo Fischione, and Vahid Tarokh. 2018. Communication Complexity of Dual Decomposition Methods for Distributed Resource Allocation Optimization. *IEEE Journal of Selected Topics in Signal Processing* 12, 4 (2018), 717–732.
- [53] Sindri Magnússon, Chinwendu Enyioha, Na Li, Carlo Fischione, and Vahid Tarokh. 2018. Convergence of limited communication gradient methods. *IEEE Journal of Selected Topics in Signal Processing* 63, 5 (2018), 1356–1371.
- [54] Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. 2012. Converting Online Algorithms to Local Computation Algorithms. In *Proc. of 39th Intl. Colloq. on Automata, Lang. and Prog. (ICALP)*. 653–664.
- [55] Laurent Massoulié and James Roberts. 1999. Bandwidth sharing: objectives and algorithms. In *IEEE INFOCOM*, Vol. 3. 1395–1403.
- [56] Brendan McMahan and Daniel Ramage. Accessed: 2017-04-10. Federated learning: Collaborative machine learning without centralized training data. <https://research.googleblog.com/2017/04/federated-learning-collaborative.html>.

- [57] Damon Mosk-Aoyama, Tim Roughgarden, and Devavrat Shah. 2010. Fully distributed algorithms for convex optimization problems. *SIAM J. on Opt.* 20, 6 (2010), 3260–3279.
- [58] João F. C. Mota, João M. F. Xavier, Pedro M. Q. Aguiar, and Markus Püschel. 2013. D-ADMM: A communication-efficient distributed algorithm for separable optimization. *IEEE Trans. on Sig. Proces.* 61, 10 (2013), 2718–2723.
- [59] Angelia Nedic and Asuman Ozdaglar. 2007. On the Rate of Convergence of Distributed Subgradient Methods for Multi-agent Optimization. In *Decision and Control (CDC), 2007 IEEE 46th Annual Conference on*. IEEE, 4711–4716.
- [60] Angelia Nedic and Asuman Ozdaglar. 2009. Distributed subgradient methods for multi-agent optimization. *IEEE Trans. on Autom. Control* 54, 1 (2009), 48–61.
- [61] Angelia Nedić and Asuman Ozdaglar. 2010. Convergence rate for consensus with delays. *Journal of Global Optimization* 47, 3 (2010), 437–456.
- [62] Angelia Nedic, Asuman Ozdaglar, and Pablo A. Parrilo. 2010. Constrained Consensus and Optimization in Multi-Agent Networks. *IEEE Trans. Automat. Control* 55, 4 (2010).
- [63] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*.
- [64] Reza Olfati-Saber. 2007. Distributed Kalman filtering for sensor networks. In *Proc. of IEEE CDC*. 5492–5498.
- [65] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. 2002. Distributing streaming media content using cooperative networking. In *Proceedings of workshop on Network and operating systems support for digital audio and video*. ACM, 177–186.
- [66] Daniel P. Palomar and Mung Chiang. 2007. Alternative distributed algorithms for network utility maximization: Framework and applications. *IEEE Trans. on Autom. Control* 52, 12 (2007), 2254–2269.
- [67] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I. Jordan, Kannan Ramchandran, Chris Re, and Benjamin Recht. 2016. CYCLADES: Conflict-free Asynchronous Machine Learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*.
- [68] Mark A. Paskin, Carlos E. Guestrin, and Jim McFadden. 2005. A robust architecture for distributed inference in sensor networks. In *Proceedings of the 4th ACM IPSN*.
- [69] David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications.
- [70] Qiuyu Peng and Steven H. Low. 2016. Distributed optimal power flow algorithm for radial networks, I: Balanced single phase case. *IEEE Transactions on Smart Grid* (2016).
- [71] Robin L Raffard, Claire J Tomlin, and Stephen P Boyd. 2004. Distributed optimization for cooperative agents: Application to formation flight. In *Proc. of IEEE Conference on Decision and Control*, Vol. 3. 2453–2459.
- [72] Pradeep Ravikumar, Alekh Agarwal, and Martin J. Wainwright. 2010. Message passing for graph-structured linear programs: Proximal methods and rounding schemes. *JMLR* 11 (2010), 1043–1080.
- [73] Omer Reingold and Shai Vardi. 2016. New techniques and tighter bounds for local computation algorithms. *Journal of Computer and System Science* 82, 7 (2016), 1180–1200.
- [74] Ralph Tyrrell Rockafellar. 1984. *Network Flows and Monotropic Optimization*. John Wiley and Sons, New York.
- [75] Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. 2011. Fast Local Computation Algorithms. In *Proc. 2nd Sym. on Innov. in Computer Science (ICS)*. 223–238.
- [76] Michael Saks and C. Seshadhri. 2010. Local Monotonicity Reconstruction. *SIAM J. on Comp.* 39, 7 (2010), 2897–2926.
- [77] Pedram Samadi, Amir-Hamed Mohsenian-Rad, Robert Schober, Vincent WS Wong, and Juri Jatskevich. 2010. Optimal real-time pricing algorithm based on utility maximization for smart grid. In *Proc. of IEEE Smart Grid Communications (SmartGridComm)*. 415–420.
- [78] Sujay Sanghavi, Dmitry M. Malioutov, and Alan S. Willsky. 2008. Linear programming analysis of loopy belief propagation for weighted matching. In *Proc. of NIPS*. 1273–1280.
- [79] Ioannis D. Schizas, Alejandro Ribeiro, and Georgios B. Giannakis. 2008. Consensus in ad hoc WSNs with noisy links—Part I: Distributed estimation of deterministic signals. *IEEE Trans. on Signal Processing* 56, 1 (2008), 350–364.
- [80] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. 2011. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming* 127, 1 (2011), 3–30.
- [81] Wei Shi, Qing Ling, Gang Wu, and Wotao Yin. 2015. A Proximal Gradient Algorithm for Decentralized Composite Optimization. *IEEE Transactions on Signal Processing* 63, 22 (2015).
- [82] Naum Z. Shor. 2012. *Minimization methods for non-differentiable functions*. Vol. 3. Springer Science & Business Media.
- [83] Rayadurgam Srikant. 2012. *The mathematics of Internet congestion control*. Springer Science & Business Media.
- [84] Gabriele Steidl and Tanja Teuber. 2010. Removing multiplicative noise by Douglas-Rachford splitting methods. *Journal of Math. Imaging and Vision* 36, 2 (2010), 168–184.
- [85] Ichiro Suzuki and Masafumi Yamashita. 1999. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.* 28, 4 (1999), 1347–1363.

- [86] Håkan Terelius, Ufuk Topcu, and Richard M. Murray. 2011. Decentralized multi-agent optimization via dual decomposition. *IEEE Trans. Automat. Control* 44, 1 (2011).
- [87] John N. Tsitsiklis, Dimitri P. Bertsekas, and Michael Athans. 1986. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Trans. on Autom. Control* 31 (1986), 803–812.
- [88] Ermin Wei, Asuman Ozdaglar, and Ali Jadbabaie. 2015. A distributed Newton method for network utility maximization: Algorithm. *IEEE Trans. on Autom. Control* 58, 9 (2015), 2162–2175.
- [89] David P. Woodruff. 2014. Sketching as a Tool for Numerical Linear Algebra. *Found. and Trends in Theoretical Computer Science* 10, 1-2 (2014), 1–157.
- [90] Yung Yi and Mung Chiang. 2008. Stochastic network utility maximization – a tribute to Kelly’s paper published in this journal a decade ago. *European Transactions on Telecommunications* 19, 4 (2008), 421–442.
- [91] Ruiliang Zhang and James T. Kwok. 2014. Asynchronous Distributed ADMM for Consensus Optimization. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML’14)*. 1701–1709.

Received August 2019; revised September 2019; accepted October 2019