A Flexible Distributed Optimization Framework for Service of Concurrent Tasks in Processing Networks

Zai Shi[1] and Atilla Eryilmaz[1]

[1]The Ohio State University, Columbus, OH 43210, USA
[1]Email:{shi.960, eryilmaz.2}@osu.edu

***Abstract*—Distributed optimization has important applications in the practical implementation of machine learning and signal processing setup by providing means to allow interconnected network of processors to work towards the optimization of a global objective with intermittent communication. Existing works on distributed optimization predominantly assume all the processors storing related data to perform updates for the optimization task in each iteration. However, such optimization processes are typically executed at shared computing/data centers along with other concurrent tasks. Therefore, it is necessary to develop efficient distributed optimization methods that possess the flexibility to share the computing resources with other ongoing tasks. In this work, we propose a new first-order framework that allows for this flexibility through a probabilistic computing resource allocation strategy while guaranteeing the satisfactory performance of distributed optimization. Our results, both analytical and numerical, show that by controlling a flexibility parameter, our suite of algorithms (designed for various scenarios) can achieve the lower computation and communication costs of distributed optimization than their inflexible counterparts. This framework also enables the fair sharing of the common resources with other concurrent tasks being processed by the processing network.**

I. Introduction

Distributed optimization concerns the minimization of the average $\frac{1}{n}\sum_{i=1}^{n} f_i(x)$ of functions f_i, each of which is only accessible by a unique processor in a network. Processors work with their own functions and communicate with their neighbors to find the optimum of the above global objective. This setup has found many applications in machine learning and signal processing fields. For example, when faced with a big dataset, we often divide it to several small datasets and process them in different servers connected by a network, such as MapReduce scheme [3]. If the problem is empirical risk minimization [2] for supervised learning, then f_i represents the loss function of the local dataset in Server i and each processor needs to determine the parameter of the prediction function that minimizes the average loss over the entire dataset. This process can be formulated as distributed optimization. Another example is decentralized estimation. In a wireless sensor network, each sensor has a measurement of the parameter that we are interested in. Each measurement contains noise from the environment and the sensor itself. How to estimate the parameter based on all the measurements is also a distributed optimization problem.

This paper is funded primarily by the NSF grant CMMI-SMOR-1562065, and in part by: the NSF grants: CCSS-EARS-1444026, CNS-NeTS-1514260, CNS-NeTS-1717045, CNS-ICN-WEN-1719371, and CNS-SpecEES-1824337; and the DTRA grants: HDTRA1-15-1-0003; HDTRA1-18-1-0050.

One of the pioneering works on distributed optimization was Tsitsiklis et al.'s work [22]. Since then, several types of methods have been proposed in this area, such as distributed subgradient descent (DSD) [8], [14], distributed dual averaging [4], [21], Alternating Direction Method of Multipliers (ADMM) [9], [18], Nesterov's method [15], [17] and second-order algorithm [10], [23], with different performances and restrictions. Among these types, DSD is the most important algorithm because it is easily implemented in a distributed way (ADMM needs sequential variable updates and second order methods need costly distributed Hessian calculation), and the basis of many further developed algorithms. For example, by adding history gradient information to DSD, the methods in [13] and [16] can achieve a linear convergence rate for the sum of strongly convex and smooth functions with a constant stepsize. Nesterov's method can also be considered as a variant of the gradient method. So in this paper, we will focus on gradient-based algorithms.

Meanwhile, we notice that prior works predominantly consider the scenario of *simultaneous updates of all network processors* for single distributed optimization task. However, in real processing networks, multiple tasks must typically be performed simultaneously by the same computing resources. Consequently, the computing resources must necessarily be shared amongst concurrent tasks including distributed optimization for satisfactory performance, which is largely overlooked in prior works.

This motivates us in this work to propose a set of flexible distributed subgradient algorithms that allow processors to work on multiple ongoing tasks simultaneously by probabilistically switching between them. Our contributions along with the organization of our paper can be summarized as follows:

- We introduce a randomization parameter into the distributed optimization framework that allows sharing of resources amongst concurrent tasks being processed by the network. This mechanism provides the necessary flexibility without disturbing the distributed nature of the optimization. The details will be presented in Section II

- We analyze the performance of our randomized distributed optimization framework for various classes of functions. Our investigations show that for various classes of functions, the convergence and convergence rate characteristics of the traditional DSD [14] are maintained while parallel processing ability can be achieved with lower computation costs in our framework. The discussion will be presented in Section III.
- Moreover, we develop a variant of our basic algorithm to further reduce the communication cost of our basic algorithm in situations where communication costs are non-negligible with respect to computation costs. This variant, called PUSD with less communication, will be introduced in Section IV.
- Through the experiments of two types of regression problems in Section V, the efficiency and the fairness of our algorithms are clearly demonstrated when compared with some traditional and newly-proposed schemes.

As an interesting related work, Nedić [11] proposes a distributed subgradient method whereby only one processor is randomly chosen to broadcast its information to its neighbors. While this work includes asynchronous update of processors, it sill lacks the flexibility of choosing how much computing resource will be put on distributed optimization. In contrast, our algorithm overcomes this disadvantage by employing a probabilistic computation strategy that is flexible enough to accommodate all degrees of updates between the extremes of the vanishingly rare and the full simultaneous updates. We also provide convergence rates of our algorithms for several classes of functions, which are not presented in [11].

Notation. Throughout the rest of the paper, n is the number of processors in the network. In the algorithms, $x_i^k, u_i^k, y_i^k, z_i^k$ are the values of Processor i in iteration k and a_{ij}^k is the weight which processor i puts on the value sent from Processor j. $\partial f(x)$ refers to the set of subgradients of the function f at x and $\nabla f(x)$ is the gradient of f at x. $||\cdot||_p$ denotes the l_p-norm for vectors. If A is a matrix, then A_{ij} refers to the (i,j)th entry of A. A^T is defined as the transpose of A. $\pi_\chi\{\cdot\}$ is the projection operator defined as

$$\pi_\chi\{x\} = \arg\min_{y\in\mathcal{X}} ||y-x||_2^2$$

Other notations are defined once mentioned in the paper.

II. Model and Basic Algorithm

We consider the following optimization problem processed by a network consisting of n processors:

$$\min_{x\in\chi} f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{1}$$

where $f_i(x)$ is convex over χ for all i, $\chi \subseteq \mathbb{R}^d$ is a convex set. The network is represented by a static connected undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Processor i can only have access to its own local function f_i and compute its (sub)gradients. Meanwhile, Processor i can communicate its values with its neighbors, the set of which is denoted as $N(i)$. We assume that the computation cost of computing a subgradient is 1 and the communication cost of one link between two processors is τ, which can be larger or smaller than 1.

In Section I, we already mentioned the application of distributed optimization. Here we take one example to explain why we are interested in solving (1) within our model. In supervised learning setup, we consider the situation where the training dataset is stored separately in several servers interconnected by a network. If each server wants to solve the empirical risk minimization problem based on its stored training dataset, then we can write $f_i(\theta) = l(h(x_i;\theta), y_i)$ where $h(\cdot)$ is the prediction function such as a linear model [6], $l(\cdot)$ is the loss function such as l_2 norm loss [6], (x_i, y_i) is the training dataset stored in Server i and θ is the global variable that the servers try to optimize. The detailed setup will be shown in Section V.

We assume that (1) has an nonempty optimality set S and denote $x^* \in S$. Meanwhile, we give the following assumptions about $f_i(x)$, some of which may be used in the next section. Note that these assumptions are standard in optimization literature [1].

Assumption 1. *$f_i(x)$ has bounded subgradients for any i and $x \in \chi$, i.e., $||g_i||_2 \le C$ for some constant C, where $g_i \in \partial f_i(x)$.*

Assumption 2. *$f_i(x)$ is L-smooth for any i and $x \in \chi$, i.e., $||\nabla f_i(x) - \nabla f_i(y)||_2 \le L||x-y||_2$, where $x, y \in \chi$.*

Assumption 3. *$f_i(x)$ is μ-strongly convex for any i and $x \in \chi$, i.e., $f_i(y) \ge f_i(x) + g_i^T(y-x) + \frac{\mu}{2}||y-x||_2^2$, where $g_i \in \partial f_i(x)$ and $x, y \in \chi$.*

The goal of distributed optimization is to make the value x_i^k of each processor converge to the optimal solution of (1) using limited communication with neighbors. Previous methods predominantly focus on solving this problem where each processor updates its value concurrently in each iteration. One of the questions we address in this work is whether it is possible to make a subset of the processors update the values in each iteration while still guaranteeing the convergence of the algorithm. Such a partial update may potentially reduce the computation cost of the network supposing each processor is doing multi-tasks. When an processor is not involved in the process of solving (1) in this iteration, it can spare its resource in other tasks, such as measurement, data preprocessing or other distributed optimization problems.

Inspired by the above motivation and the stochastic gradient descent (SGD) [2], we propose Partially Updated Subradient Descent (PUSD) shown in Algorithm 1, which can be implemented distributedly in each processor. This algorithm can be divided into two operations. Steps 6 and 7 refer to the communication operation, where each processor communicates with its neighbors to average the information from its own and the neighbors. Steps 8 to 14 refer to the computation operation, where a subset of processors compute the subgradients and run the subgradient descent to approach the optimal solution of (1). The other processors just maintain their original values

Algorithm 1 Partially Updated Subgradient Descent (PUSD) for Processor i

1: **Input:** number of iterations K, probability to run subgradient descent p, stepsizes for K iterations $\{\eta_k\}_{k=1}^K$.
2: **Output:** Option I: x_i^K; Option II: $v_i^K := \frac{\sum_{k=1}^K \eta_k x_i^k}{\sum_{k=1}^K \eta_k}$; Option III: $z_i^K := \frac{2}{K(K+1)} \sum_{k=1}^K k x_i^k$
3: Initialize $x_i^0 \in \chi$.
4: Set $u_i^0 \leftarrow x_i^0$.
5: **for** $k = 1$ **to** K **do**
6: Send u_i^{k-1} to the neighbor processors and receive $\{u_j^{k-1}\}_{j \in N(i)}$ from all the neighbor processors.
7: Set $x_i^k \leftarrow a_{ii}^k u_i^{k-1} + \sum_{j \in N(i)} a_{ij}^k u_j^{k-1}$ for some $a_{ii}^k \in [0,1], a_{ij}^k \in [0,1]$.
8: Generate a random number r in $[0,1]$.
9: **if** $r < p$ **then**
10: Compute a subgradient g_i^k of $f_i(x_i^k)$.
11: Set $u_i^k \leftarrow \pi_\chi\{x_i^k - \eta_k g_i^k\}$.
12: **else**
13: Set $u_i^k \leftarrow x_i^k$ and process other tasks.
14: **end if**
15: **end for**

and process different tasks. Using this algorithm, we can process several tasks concurrently in this network with different emphasis on distributed optimization adjusted by p in Algorithm 1. The traditional DSD [14] corresponds to our algorithm when $p = 1$. In this way, PUSD can be regarded as a generalization of the traditional DSD by providing the flexible parallel processing ability.

Meanwhile, there exists some hard-partitioning schemes, such as assigning a partition of processors to a certain task, which also allows for parallel processing. But we should mention that the distributed optimization framework is especially important under many circumstances whereby different processors have access to chunks of data that may be private or infeasible to share with other processors in the network. The example of wireless sensor networks mentioned in Section I is such a case where each measurement is related to its location and not easy to be shared. The aim of the probabilistic switching is to make the gradient computed in each step a good estimation of the full gradient of the global function, which is essential for the convergence results shown in the next section. Hard-partitioning schemes over network processors may be difficult to realize this end, especially under the above-mentioned circumstances. Other alternatives such as time division multiplexing can be regarded as specific implementations of sequential processing, which has been compared with our methods in Section V.

In the next section, we will show how the probabilistic switching impacts convergence results of distributed optimization problems.

III. Convergence Results for PUSD

First we give some definitions prepared for an assumption related to the communication operation of PUSD. Define $A(k) \in \mathbb{R}^{n \times n}$ as the weight matrix in iteration k whose (i,j)th entry is a_{ij}^k. Obviously $a_{ij}^k \neq 0$ only when edge $(i,j) \in \mathcal{E}$ or $j = i$. Supposing $\{A(k)\}_{k=1,2,\ldots}$ are independent and identically distributed (i.i.d), we can define $\bar{A} \triangleq \mathbb{E}[A(k)]$ for all $k > 0$. With the edge set induced by the positive elements of $\bar{A}$, i.e.,

$$\bar{\mathcal{E}} = \{(j,i) | \bar{A}_{ij} > 0\}, \tag{2}$$

we define $\bar{G} = (\mathcal{V}, \bar{\mathcal{E}})$ as the mean connectivity graph, where $\mathcal{V}$ is the vertex set of the original network.

Now we give the following the assumption about $A(k)$:

Assumption 4. *(a) $\{A(k)\}_{k=1,2,\ldots}$ are i.i.d.;*
(b) There exists some constant $\gamma \in (0,1)$ such that $A(k)_{ii} \geq \gamma$ with probability 1 for all i and $\min_{j,i \in \bar{\mathcal{E}}} \frac{\bar{A}_{ij}}{2} \geq \gamma$;
(c) The mean connectivity graph is strongly-connected;
(d) $A(k)$ is doubly stochastic with probability 1 for all $k > 0$, i.e., $\sum_i a_{ij}^k = \sum_j a_{ij}^k = 1$ for any i and j.

Remark 1. *A simple weight matrix satisfying Assumption 4 is called the lazy Metropolis update [12], which is defined as follows:*

$$a_{ij}^k = \begin{cases} \frac{1}{2\max\{d_i, d_j\}} & \text{if } j \in N(i) \\ 1 - \sum_{j \in N(i)} \frac{1}{2\max\{d_i, d_j\}} & \text{if } j = i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

for all $k > 0$, where d_i is the degree of Processor i. In this case $A(k)$ is deterministic. The mean connectivity graph is strongly-connected if and only if the underlying graph of the network is strongly-connected. Using this matrix requires each processor to broadcast its degree to its neighbors along with its value.

Now we will give three theorems related to the convergence results of PUSD. Due to the space constraint, their proofs will be presented in the form of outlines. The details can be found at [19]. Here we define two values used in the theorems: $B = (3 + \frac{2}{\gamma^{2(n-1)}}) \exp\{-\frac{\gamma^{4(n-1)}}{2}\}$ and $\theta = \exp\{-\frac{\gamma^{4(n-1)}}{4(n-1)}\}$, where γ is defined in Assumption 4.

Theorem 1. *If Assumption 1 and Assumption 4 are satisfied, then for Algorithm 1 with output Option II we have:*
(a) when $\eta_k = \eta = \frac{1}{\sqrt{K}}$[1] is fixed, where K is a constant,

$$\mathbb{E}[f(v_i^K) - f(x^*)] \leq \frac{1}{\sqrt{K}} \left(\frac{||y^0 - x^*||_2^2}{2p} + \frac{2(1 - p + np + \frac{1}{2}n)C^2}{n} + \frac{8npBC^2}{1-\theta} \right) \tag{4}$$

[1] We can choose $\eta_k = \frac{\alpha}{\sqrt{K}}$ for any $\alpha > 0$. Here we use $\frac{1}{\sqrt{K}}$ for the ease of discussion

for any i, where $y^0 = \frac{1}{n}\sum_{i=1}^n x_i^0$;
(b) when $\sum_{k=1}^\infty \eta_k = \infty$ and $\sum_{k=1}^\infty \eta_k^2 < \infty$,

$$\lim_{K\to\infty} \mathbb{E}[f(v_i^K) - f(x^*)] = 0 \quad (5)$$

for any i.

Proof outline. 1. Write one step of Algorithm 1 in a compact form:

$$y^{k+1} = y^k + \frac{1}{n}\sum_{i\in I_k}(\Delta_i^k - \eta_k g_i^k) \quad (6)$$

where $y^{k+1} = \frac{1}{n}\sum_{i=1}^n x_i^{k+1}$, $v_i^k = x_i^k - \eta_k g_i^k$, $\Delta_i^k = \pi_\chi\{v_i^k\} - v_i^k$ and I_k is the set of processors computing (sub)gradients in iteration k.
2. Expand $\mathbb{E}_{I_k}[||y^{k+1}-x^*||_2^2|\{x_i^k\}_{i=1}^n]$ by plugging (6) and get the following upper bound using the definition of subgradients, the triangle inequality, the Cauchy-Schwartz inequality, Assumption 1 and the projection theorem $(v_i^k - \pi_\chi\{v_i^k\})^T(x^* - \pi_\chi\{v_i^k\}) \le 0$ [1]:

$$\mathbb{E}_{I_k}[||y^{k+1}-x^*||_2^2|\{x_i^k\}_{i=1}^n] \le ||y^k - x^*||_2^2 + \frac{4\eta_k^2 p(1-p)C^2}{n} + 4\eta_k^2(p^2+\frac{1}{2}p)C^2 + 8p\eta_k C D_k - 2p\eta_k(f(x_j^k) - f(x^*)) \quad (7)$$

where $D_k = \max_{i\in\{1,2,\ldots,n\}} ||y^k - x_i^k||_2$.
3. Take expectations over all the previous values and bound $\mathbb{E}D_k$ using the similar method in Lemma 7 of [8]. With the bound $\mathbb{E}D_k \le np\sum_{r=1}^k \eta_{r-1}CB\theta^{k-r}$ where $\eta_0 = 1$, we have

$$\mathbb{E}[||y^{k+1}-x^*||_2^2] \le \mathbb{E}||y^k - x^*||_2^2 + \frac{4\eta_k^2 p(1-p)C^2}{n} + 4\eta_k^2(p^2+\frac{1}{2}p)C^2 + 8np^2\eta_k C^2\sum_{r=1}^k \eta_{r-1}B\theta^{k-r} - 2p\eta_k\mathbb{E}[f(x_j^k) - f(x^*)] \quad (8)$$

4. Telescope from $k = 1$ to K, and rearrange the terms to get the upper bound of $\mathbb{E}[\frac{\sum_{k=1}^K \eta_k(f(x_j^k)-f(x^*))}{\sum_{k=1}^K \eta_k}]$. Use the assumptions of η_k and the convexity of f to obtain the final results for case (a) and (b) respectively. □

Remark 2. *The last term of equation* (4) *are called consensus error because it is brought by the processor disagreement after K iterations. We can see that it is determined by the weight matrix and the topology of the network via B and θ.*

In Theorem 1, if we let the right side of (4) equal to ε and solve K from the equation, then we can get the the maximum iterations to achieve ε-suboptimality, which is written as (we allow K to be non-integer for simplicity):

$$K = \frac{1}{\varepsilon^2}(\frac{||y^0 - x^*||_2^2}{2p} + \frac{2(1-p+np+\frac{1}{2}n)C^2}{n} + \frac{8npBC^2}{1-\theta})^2$$

On average, np processors are computing the gradients in each iteration. Define the network computation cost of distributed optimization as the sum of the computation cost brought by the processors which compute the subgradients in the network. Then the expected network computation cost to achieve ε-suboptimality is

$$\text{Ecomp}(\varepsilon) \le \frac{np}{\varepsilon^2}(\frac{1}{2p}||y^0 - x^*||_2^2 + \frac{2(1-p+np+\frac{1}{2}n)C^2}{n} + \frac{8npBC^2}{1-\theta})^2 \quad (9)$$

where $p \in (0, 1]$. Since the traditional DSD corresponds to our algorithm with $p = 1$, then its expected network computation cost is

$$\text{Ecomp}(\varepsilon) \le \frac{n}{\varepsilon^2}(\frac{1}{2}||y^0 - x^*||_2^2 + 3C^2 + \frac{8nBC^2}{1-\theta})^2 \quad (10)$$

which can be also obtained following the proof line in [8]. Now we fix B and θ by using the same weight matrix for different p. Then from (9) and (10) we know that when C is large, our algorithm can be better than the traditional DSD in terms of the expected network computation cost of distributed optimization for some p. If we know the relation between the computation cost of concurrent tasks and the number of their operating processors, we can further optimize the whole computation cost by minimizing the cost function with an appropriate p.

Theorem 2. *If Assumption 1 ,3 and 4 are satisfied, then for Algorithm 1 with output Option III and the diminishing stepsize $\eta_k = \frac{2}{\mu p(k+1)}$,*

$$\mathbb{E}[f(z_i^K) - f(x^*)] \le \frac{1}{K+1}(\frac{8(1-p+np+\frac{1}{2}n)C^2}{n\mu p} + \frac{16nBC^2}{(1-\theta)\mu} + \frac{8npBC^2}{K(1-\theta)}) \quad (11)$$

$$\xrightarrow{K\to\infty} 0$$

Proof outline. Step 1, 2 and 3 are similar to the proof of Theorem 1 except taking strong convexity into consideration by using the property in Assumption 3. Then the bound in (8) changes to

$$\mathbb{E}(f(x_j^k) - f(x^*)) \le \frac{1}{2p\eta_k}(1-p\mu\eta_k)\mathbb{E}||y^k - x^*||_2^2 - \frac{1}{2p\eta_k}\mathbb{E}||y^{k+1}-x^*||_2^2 + \frac{2\eta_k((1-p)+np+\frac{1}{2}n)C^2}{n} + 4npBC^2\sum_{r=1}^k \eta_{r-1}\theta^{k-r}$$

4. Multiply both sides by k and telescope from $k = 1$ to K. Use $\eta_k = \frac{2}{\mu p(k+1)}$ to let some of the terms cancel each other and bound the term $\sum_{k=1}^K k\sum_{r=1}^k \eta_{r-1}\theta^{k-r}$. Then we have

$$\sum_{k=1}^K k\mathbb{E}(f(x_j^k) - f(x^*)) \le -\frac{\mu K(K+1)}{4}\mathbb{E}||y^{K+1} - x^*||_2^2 + \frac{4K(1-p+np+\frac{1}{2}n)C^2}{n\mu p} + \frac{8(K-1)nBC^2}{(1-\theta)\mu} + \frac{4npBC^2}{1-\theta}$$

5. Multiplying both sides by $\frac{2}{K(K+1)}$ and using the convexity of f gives the final result. □

From Theorem 2, we can see that our algorithm has a convergence rate of $O(1/K)$, which matches the order of the convergence rate of centralized stochastic gradient descent [2]. When K is large, the last term of (11) can be neglected. Then using the same procedure of deriving (9), we can write the bound of the expected network computation cost to reach ε-suboptimality as

$$\begin{aligned}&\text{Ecomp}(\varepsilon)\\&\le \frac{np}{\varepsilon}\left(\frac{8(1-p+np+\frac{1}{2}n)C^2}{n\mu p}+\frac{16nBC^2}{(1-\theta)\mu}\right)-np \end{aligned} \quad (12)$$

in this case. Again, when C is large, the expected network computation cost of distributed optimization for PUSD can be lower than the traditional DSD by controlling p.

Theorem 3. *If Assumption 1, 2, 3 and 4 are satisfied, then for Algorithm 1 with output Option I and the fixed stepsize* $\eta_k=\eta\in(\frac{L-\mu}{p\mu L},\frac{1}{p\mu})$,

$$\begin{aligned}\mathbb{E}[f(x_i^{K+1})-f(x^*)]\le (\frac{L}{\mu}-pL\eta)^K(f(y^0)-f(x^*)-R)\\ +R+\frac{np\eta BC^2}{1-\theta}+npC^2B\theta^{K-1}\end{aligned} \quad (13)$$

$$\xrightarrow{K\to\infty} R+\frac{np\eta BC^2}{1-\theta} \quad (14)$$

for any i, *where* $y^0=\frac{1}{n}\sum_{i=1}^n x_i^0$, *and*

$$R=\frac{\frac{2\eta^2L(p(1-p)+np^2+\frac{1}{2}np)C^2}{n}+\frac{3np^2LBC^2(\eta^2+\eta)}{1-\theta}}{1-\frac{L}{\mu}+pL\eta}.$$

Proof outline. Since f is μ-strongly convex, L-smooth, and $\nabla f(x^*)=0$, we have [1]

$$\frac{1}{2}\mu||y-x^*||_2^2\le f(y)-f(x^*)\le\frac{1}{2}L||y-x^*||_2^2$$

Then the bound in (8) can be improved to

$$\begin{aligned}&\mathbb{E}[f(y^{k+1})-f(x^*)]\le(\frac{L}{\mu}-pL\eta_k)\mathbb{E}(f(y^k)-f(x^*))\\&+\frac{2\eta_k^2L(p(1-p)+np^2+\frac{1}{2}np)C^2}{n}\\&+3np^2LBC^2\eta_k\sum_{r=1}^{k}\eta_{r-1}\theta^{k-r}\end{aligned}$$

When $\eta_k=\eta$ is fixed, we have

$$\begin{aligned}\mathbb{E}[f(y^{k+1})-f(x^*)]-R&\le(\frac{L}{\mu}-pL\eta)(\mathbb{E}[f(y^k)-f(x^*)]-R)\\&\le(\frac{L}{\mu}-pL\eta)^k(f(y^0)-f(x^*)-R)\end{aligned}$$

where R is defined in Theorem 3. If $\eta\in(\frac{L-\mu}{p\mu L},\frac{1}{p\mu})$, then $(\frac{L}{\mu}-pL\eta)\in(0,1)$. By incorporating the consensus error, we have

$$\begin{aligned}\mathbb{E}[f(x_i^{k+1})-f(x^*)]\le(\frac{L}{\mu}-pL\eta)^k(f(y^0)-f(x^*)-R)\\+R+\frac{np\eta BC^2}{1-\theta}+npC^2B\theta^{k-1}\\ \xrightarrow{k\to\infty} R+\frac{np\eta BC^2}{1-\theta}\end{aligned}$$

□

From Theorem 3, we can see that a part of the optimality gap decreases in a linear rate. When this part dominates the whole optimality gap, the algorithm "seems" to have a linear convergence rate, which can be observed in Section V-B.

IV. PUSD WITH LESS COMMUNICATION

For the purpose of simplicity, the fixed weight matrix like (3) is often used in distributed optimization methods, where all the processors are involved in the communication operation. However, this may not be efficient when the communication cost is very large. This situation may arise in the example of wireless sensor networks when some channels are in deep fading [20]. Assumption 4 allows us to choose some suitable weight matrix to reduce the communication cost by not utilizing all the links, but we need additional communication protocols to realize such a weight matrix. In our distributed optimization framework, this aim can be reached directly by taking advantage of the randomness brought by p in PUSD. The modified algorithm, called PUSD with less communication, is shown in Algorithm 2.

Here we call the processors which compute their subgradients the *active* processors and the others the *inactive* processors. Note that the inactive processors are actually processing other tasks and their inactivity is only with respect to the distributed optimization being performed. Define $N'(i)$ as the set of active neighbors for the inactive processors. In this algorithm, the communication only occurs between the active processors and their neighbors. For example, suppose Algorithm 2 is running in a network shown in Figure 1. If Processor 3 and Processor 5 choose to run the subgradient descent in this iteration, then the communication will happen in the links represented by bold lines in Figure 1. A time window can be set for the inactive processors to guarantee the receipt of messages from all the active neighbors.

If $A(k)$ is doubly stochastic with probability 1 for all k and the network is strongly connected, it is easy to check that Assumption 4 is satisfied for Algorithm 2. Compared with Algorithm 1, we exchange the order of the communication operation and the computation operation. In this way we can guarantee the same convergence results in Section III using the same line of argument, which is explained in the proof of the following theorem.

Theorem 4. *For Algorithm 2, Theorem 1, 2 and 3 still hold with their respective assumptions.*

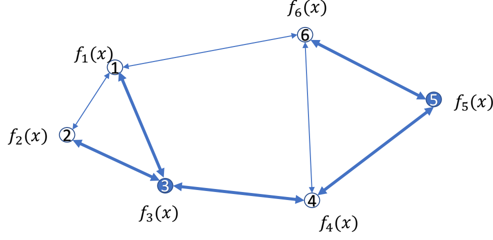


Fig. 1. The communication operation of Algorithm 2 in a network consisting of 6 processors in one iteration. The colored processors are active ones in this iteration and the bold lines are the links involved in the communication operation.

Proof. The proof of this theorem is similar to Theorem 1, 2 and 3. We focus on the explanation of the reason why we should exchange the order of the communication operation and the computation operation.

1. Write one step of Algorithm 2 in a compact form:

$$y^{k+1} = y^k + \frac{1}{n} \sum_{i \in I_{k+1}} (\Delta_i^k - \eta_k g_i^k) \tag{15}$$

where $y^{k+1} = \frac{1}{n}\sum_{i=1}^n x_i^{k+1}$, $v_i^k = x_i^k - \eta_k g_i^k$, $\Delta_i^k = \pi_\chi\{v_i^k\} - v_i^k$ and I_{k+1} is the set of processors computing gradients in iteration $k+1$.

2. The main difference is in this step. Here we will take expectations to I_{k+1} conditioned on $\{x_i^k\}_{i=1}^n$ and then follow the same proof line to get the same convergence results. If the order of communication operation and computation operation is not exchanged in Algorithm 2, I_k is coupled with $\{x_i^k\}_{i=1}^n$ and then we cannot obtain the bound of $\mathbb{E}_{I_k}[||y^{k+1} - x^*||_2^2 | \{x_i^k\}_{i=1}^n]$ in (7) without the independence condition.

The remaining steps are the same with Theorem 1, 2 and 3. □

As less processors are involved in the communication operation, the consensus process across the networks in Algorithm 2 may be slower than Algorithm 1. For example, in Algorithm 2 we can use a similar weight matrix to (3):

$$a_{ij}^k = \begin{cases} \frac{1}{2\max\{d_i^k, d_j^k\}} & \text{if } j \in N^k(i) \\ 1 - \sum_{j \in N^k(i)} \frac{1}{2\max\{d_i^k, d_j^k\}} & \text{if } j = i \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

where $N^k(i)$, d_i^k, d_j^k are defined with regard to $\mathcal{G}^k = (\mathcal{V}^k, \mathcal{E}^k)$. Here $\mathcal{V}^k$ is the vertex set of the active processors and their neighbors in iteration k, and $\mathcal{E}^k$ is the edge set of the edges between the active processors and their neighbors. Using this matrix, the convergence rate of Algorithm 2 may be slower than Algorithm 1 because γ in Assumption 4 may decrease, leading to the increase of B and θ in the convergence results. Therefore for specific graphs and weight matrices, there should be a communication-computation cost tradeoff in Algorithm 2. But in empirical experiments, the tradeoff may be not obvious since the theoretical results are in terms of bounds. In the Section V, we will see the examples where the computation cost has almost no change when Algorithm 2 is applied.

Algorithm 2 Partially Updated Subgradient Descent (PUSD) with Less Communication for Processor i

1: **Input:** number of iterations K, probability to run subgradient descent p, stepsizes for K iterations $\{\eta_k\}_{k=1}^K$
2: **Output:** Option I: x_i^K; Option II: $v_i^K := \frac{\sum_{k=1}^K \eta_k x_i^k}{\sum_{k=1}^K \eta_k}$; Option III: $z_i^K := \frac{2}{K(K+1)} \sum_{k=1}^K k x_i^k$
3: Initialize $x_i^0 \in \chi$.
4: **for** $k = 1$ **to** K **do**
5: Generate a random number r in $[0, 1]$.
6: **if** $r < p$ **then**
7: Compute a subgradient g_i^{k-1} of $f_i(x_i^{k-1})$.
8: Set $u_i^k \leftarrow \pi_\chi\{x_i^{k-1} - \eta_k g_i^{k-1}\}$.
9: Send u_i^k to the neighbor processors.
10: **if** received $\{u_j^k\}_{j\in N(i)}$ from all the neighbors. **then**
11: Set $x_i^k \leftarrow a_{ii}^k u_i^k + \sum_{j\in N(i)} a_{ij}^k u_j^k$ for some $a_{ii}^k \in [0,1], a_{ij}^k \in [0,1]$.
12: **end if**
13: **else**
14: Set $u_i^k \leftarrow x_i^{k-1}$ and process other tasks.
15: **if** received $\{u_j^k\}_{j\in N'(i)}$ from all the active neighbor(s). **then**
16: Send u_i^k to the active neighbor(s) and set $x_i^k \leftarrow a_{ii}^k u_i^k + \sum_{j\in N'(i)} a_{ij}^k u_j^k$ for some $a_{ii}^k \in [0,1], a_{ij}^k \in [0,1]$.
17: **end if**
18: **end if**
19: **end for**

It is insightful to consider Erdős-Renyi networks for characterizing the communication cost reduction of Algorithm 2. To that end, consider the $G(n,q)$ model where n processors connect with each other with probability q. We can choose $q = (1+\epsilon)\log(n)/n$ for some $\epsilon > 0$ to guarantee that the graph is strongly connected with high probability [5]. For this model we have the following proposition:

Proposition 1. *For Erdős-Renyi networks $G(n,q)$, the expected communication cost in each iteration of Algorithm 2 is $\frac{q}{2}(2p - p^2)(n^2 - n)\tau$.*

Proof. In Algorithm 2, a link is in communication for distributed optimization if and only if it exists and at least one of its two vertexes is active. Then

$$\mathbb{P}\{\text{a link in communication}\} = q(1 - (1-p)^2) = q(2p - p^2).$$

So the number of the links in communication follows a binomial distribution $B\left(\binom{n}{2}, 2qp - qp^2\right)$. Now we can obtain the expected communication cost in one iteration:

$$\text{Ecomm} = \frac{q}{2}(2p - p^2)(n^2 - n)\tau$$

□

In contrast, when all the processors are involved in the communication operation, the expected communication cost

will be $\binom{n}{2}q\tau$. So the decrease is $\frac{q}{2}(p-1)^2(n^2-n)\tau$. If n and τ are large, this amount can be significant.

V. Numerical Results

In this section, we will show the advantages of PUSD and its variant for linear regression problems using two loss functions, l_1-norm loss and l_2-norm loss, in the regression setting. These two objectives represent two types of functions we discussed in Section III: one satisfying Assumption 1 and one satisfying Assumption 1, 2 and 3. Therefore we can examine the performance of PUSD and its variant in a more comprehensive framework. For all the experiments, we use a network consisting of 100 processors, represented by a strongly connected 10-regular graph. Meanwhile for simplicity and practicality, we use fixed stepsizes in the experiments. Stepsizes may be chosen differently in each algorithm so that the best performance can be achieved.

A. Linear Regression: l_1-norm Loss

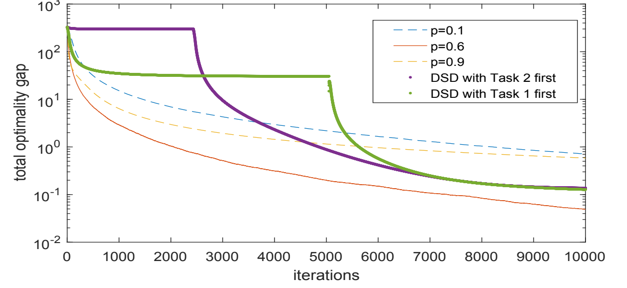
Suppose each processor has a training set consisting of 20 data points $(x_{i,m}, y_{i,m})$, where $x_{i,m} \in \mathbb{R}^d$ and $y_{i,m} \in \mathbb{R}$ belong to the mth data point stored in Processor i. Then the empirical risk minimization for the l_1-norm loss is to solve the following optimization problem:

$$\min_{\theta \in \mathbb{R}^d} f(\theta) = \frac{1}{20n}||y - X^T\theta||_1 \tag{17}$$

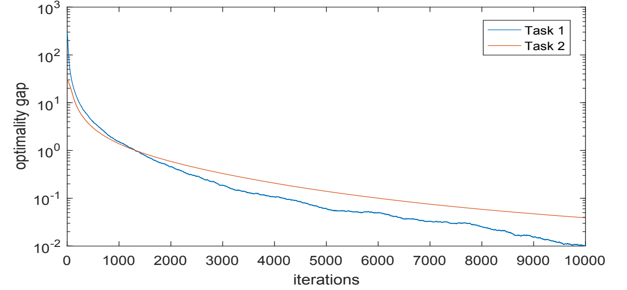
where $X \in \mathbb{R}^{d\times 20n}$ is the matrix whose $(20(i-1)+m)$th column is $x_{i,m}$ and $y \in \mathbb{R}^{20n}$ is the vector whose $(20(i-1)+m)$th entry is $y_{i,m}$. $n(=100)$ is the number of processors in the network. It is easy to check that l_1 norm functions satisfy Assumption 1.

First, we consider that two tasks are processed by the network, which both solve Problem (17) with different datasets. For Task 1, we set $y_{i,m} = x_{i,m}^T \left[\begin{smallmatrix}30\\20\\10\end{smallmatrix}\right] + \xi_{i,m}$, where $x_{i,m}$ is generated by a uniform distribution taking values in $[0,10]^3$, and $\xi_{i,m} \in \mathbb{R}$ follows a uniform distribution in $[0,5]$. For Task 2, both $x_{i,m}$ and $\xi_{i,m}$ are generated by a uniform distribution taking values in $[0,1]^3$. With different datasets, these two tasks differ in the fraction of the noise $\xi_{i,m}$ in $y_{i,m}$. When applying PUSD (Algorithm 1) to these two tasks, we let each processor process Task 1 with probability p and Task 2 otherwise, both using (3) for the weight matrix. As two sequential counterparts, the traditional DSD [14] is used to process two tasks sequentially with Task 1 first or with Task 2 first using the same weight matrix. When the optimality gap of one task is less than 10^{-1}, the network proceeds to process the other task with the DSD. [2] The optimality gap in iteration k is measured by $f(\frac{\sum_{k=1}^K \theta_i^k}{K}) - f(\theta^*)$ (Option II in Algorithm 1) where θ_i^k is the value of Processor i in iteration k. Processor i is randomly chosen at the beginning of the experiment, and kept tracked afterwards.

[2]In practice each processor has no access to the optimality gap and terminates when the value changes little. The switching criteria used here is for simplicity and comparison.



(a) The comparison of PUSD with $p = 0.1$, $p = 0.6$, $p = 0.9$ and the two sequential processing algorithms for l_1 norm loss



(b) The evolution of two tasks using PUSD with $p = 0.6$ for l_1 norm loss

Fig. 2. PUSD applied to two concurrent optimization tasks for l_1 norm loss

In Figure 2(a) we compare PUSD using different p with the two sequential counterparts. The total optimality gap is defined as the sum of the optimality gap of two tasks. From the figure we can see that $p = 0.6$ gives a faster rate than $p = 0.1$, $p = 0.9$ and the sequential counterparts. This is because when approaching the optimal point, DSD has a very slow rate if one task is processed alone. During this phase, it is unwise to still process this task using all the processors. In contrast, PUSD has a fast rate in the initial phase because of its similarity to SGD (It is the characteristic of SGD [1]). When both tasks are close to the optimum, PUSD can split the computing resources to let both tasks approach the optimum concurrently. Meanwhile, for fairness and efficiency, p should not be too extreme since we do not want one of the tasks to drag the global performance. So $p = 0.6$ gives the best result for this concurrent optimization setup. From Figure 2(b) we can also see that the optimality gaps of both tasks decrease below 10^{-1} finally when $p = 0.6$. Their curves are not apart from each other, so the fairness is maintained with this choice. This experiment shows the advantages of PUSD when dealing with concurrent optimization problems for nonsmooth functions.

In the rest of the subsection, we assume that Task 2 is processed along with other kinds of tasks. Now we apply Algorithm 2, PUSD with less communications, to Task 2 with the weight matrix defined in (16). We compare this algorithm with Algorithm 1 using (3) for different p to show how the computation cost is impacted. The measurement of the optimality gap is the same with the above experiment, but now only for Task 2.

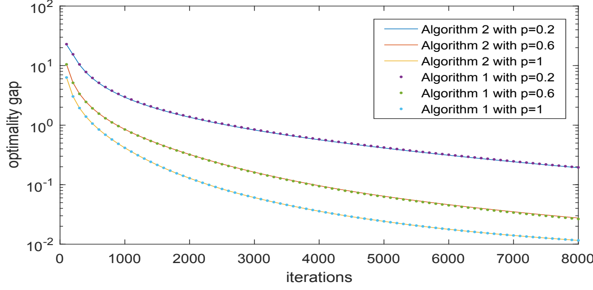


Fig. 3. The comparison of Algorithm 1 and Algorithm 2 with different p for l_1 norm loss

In Figure 3, we plot the optimality gap of Task 2 versus iterations for Algorithm 1 and Algorithm 2 when $p = 0.2$, $p = 0.6$ and $p = 1$ (which is also the traditional DSD). We can see that for the same p, the performance of the two algorithms is almost the same. So empirically, the decrease of γ in Assumption 4 does not impact the convergence rate too much when (3) and (16) are used in the 10-regular graph. In this case, Algorithm 2 can reduce the communication cost while not increasing the computation cost of Task 2. It is of significant value when the system has high communication costs. Meanwhile we can compare the computation cost of PUSD with the traditional DSD ($p = 1$) in this figure. For example, when the optimality gap is 10^{-1}, PUSD with $p = 0.6$ needs about 3900 iterations while the traditional DSD needs about 2300 iterations. So the expected network computation cost to reach 10^{-1} gap is 2.34×10^5 for PUSD with $p = 0.6$ and 2.3×10^5 for the traditional DSD, which is almost the same for two cases. But our algorithm allows for the parallel processing of other tasks with the same cost for the distributed optimization.

B. Linear Regression: l_2-norm Loss

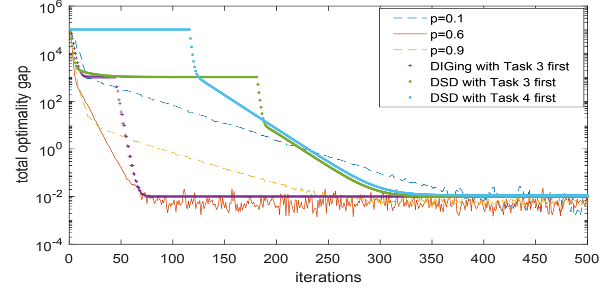
With the same definition in Subsection V-A, the empirical risk minimization for the l_2-norm loss is to solve the following optimization problem:

$$\min_{\theta \in \chi} f(\theta) = \frac{1}{20n} ||y - X^T \theta||_2^2 \tag{18}$$

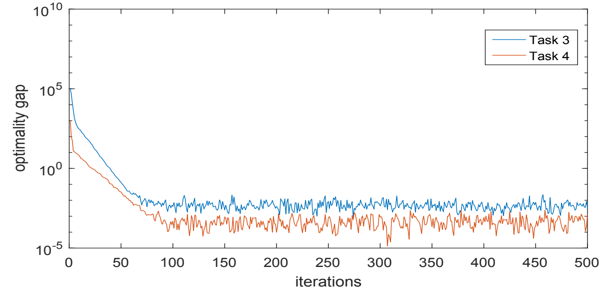
Here we assume a compact constraint $\chi = [0, 100]^3$ for θ, which makes Assumption 1, 2 and 3 all satisfied.

First, we still consider the situation where two optimization problems like (18) are being processed by the network with two datasets respectively. They are referred to as Task 3 and Task 4 in this subsection. The datasets of Task 3 are the same with Task 1 and Task 4 same with Task 2 in Subsection V-A. Here the optimality gap is measured by $f(\theta_i^k) - f(\theta^*)$ (Option I in Algorithm 1) where Processor i is still randomly chosen at the beginning of the experiment and kept tracked afterwards.

In Figure 4(a) we compare PUSD using different p with three sequential processing methods: DSD with Task 3 first, DSD with Task 4 first and DIGing algorithm [13] with Task 3 first. DIGing algorithm is a newly proposed method which



(a) The comparison of PUSD with $p = 0.1$, $p = 0.6$, $p = 0.9$ and the three sequential processing algorithms for l_2 norm loss



(b) The evolution of two tasks using PUSD with $p = 0.6$ for l_2 norm loss

Fig. 4. PUSD applied to two concurrent optimization tasks for l_2 norm loss

can achieve a linear convergence rate for the sum of strongly convex, smooth functions using a fixed stepsize. The switching criteria for sequential processing is that when the optimality gap of one task is less than 10^{-2}, the network proceeds to the next task. The metric is the total optimality gap of two tasks.

From the figure we can see that larger p gives a faster rate in the initial phase (In fact Figure 2(a) has the same trend, but not obviously). This is because Task 3 has a greater decrease in its optimality gap than Task 4 if processed alone. But when Task 3 is close to its optimum, the rate will decrease dramatically. Same with Subsection V-A, putting more weight on Task 3 is not a wise decision in this phase. So the rate of PUSD reduces after an initial phase when p is too extreme. DSD with Task 3 first and DSD with Task 4 first are also impacted by the slow final phase. Given this tradeoff, $p = 0.6$ is the best choice which only gives a little more emphasis on Task 1. For DIGing algorithm, it can obtain the exact solution in a linear rate using a fixed stepsize [13]. So it has a much better performance than DSD. But under our switching criteria where a certain optimization error is allowed, PUSD with $p = 0.6$ has a similar performance with DIGing algorithm. Given that DIGing algorithm needs to store history information and does not allow parallel processing, our algorithm has its advantages over DIGing in some applications.

In Figure 4(b), we plot the evolution of two tasks when PUSD with $p = 0.6$ is applied. From the figure, we can see that the optimality gaps of both tasks decrease similarly when $p = 0.6$, so the fairness is also guaranteed. The decrease is approximately in a linear rate before reaching the final

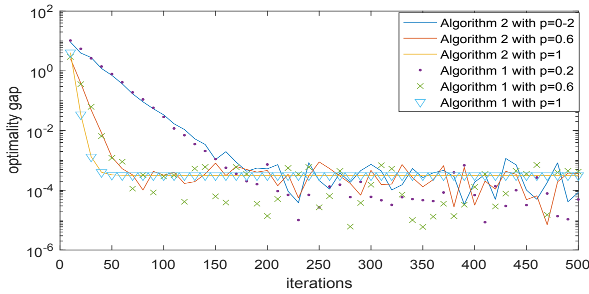


Fig. 5. The comparison of Algorithm 1 and Algorithm 2 with different p for l_2 norm loss

optimality gap, as explained after the statement of Theorem 3. This experiment shows the advantages of PUSD when dealing with concurrent optimization problems for strongly convex, smooth functions.

Now same with previous subsection, we assume that Task 4 is processed along with other kinds of tasks and focus on the performance of Task 4. We compare Algorithm 2 using the weight matrix defined in (16) with Algorithm 1 using (3) for $p = 0.2$, $p = 0.6$ and $p = 1$ (same as the traditional DSD) in Figure 5. Again, we can find that the performance of two algorithms are very close to each other for all the choices of p. In this case, Algorithm 2 is also preferable if the communication cost is non-negligible. Now we turn to the computation cost for PUSD with different p when a certain optimality gap is reached. We take 10^{-3} for example. From Figure 5, we can observe that it takes about 30, 50 and 150 iterations to reach the 10^{-3} gap for PUSD with $p = 1$, $p = 0.6$ and $p = 0.2$ respectively. It means that the expected network computation cost of distributed optimization is almost the same for these three cases. Compared with the traditional DSD, PUSD is more flexible by allowing parallel processing of other tasks with different requirements by adjusting p.

VI. Conclusion and Future Work

In this paper we proposed Partially Updated Subgradient Descent (PUSD) to enable the split of computing resources for distributed optimization and concurrent tasks in networks. We then derived the convergence results of PUSD with different assumptions on each function, which showed its favorable convergence and convergence rate characteristics. For one possible application situation, we developed PUSD with less communication to get a better performance in communication costs. The experiments of two machine learning problems demonstrated the flexibility and efficiency of our algorithms along with their ability to share computing resources more fairly amongst concurrent tasks.

There are some open problems remaining for our algorithms. First, each communication operation must wait for the completion of the computation operation in PUSD. We hope to develop an algorithm without this kind of synchronism while guaranteeing the convergence results of the original algorithm. Second, it is interesting to study whether PUSD can be accelerated for smooth functions or strongly convex, smooth functions. Because of the similarity of PUSD to SGD, variance reduction methods like SVRG [7] may be promising. But such a method needs full gradient information every several iterations, which is difficult to be implemented in our distributed setting. Thus, new techniques should be developed for this problem. Last, it is interesting to explore whether we can improve PUSD by enabling it to identify ”good” processors for each parallel task and allocate the tasks accordingly.

References

[1] D. P. Bertsekas. *Convex optimization algorithms*. Athena Scientific Belmont, 2015.

[2] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] J. C. Duchi, A. Agarwal, and M. J. Wainwright. Dual averaging for distributed optimization: Convergence analysis and network scaling. *IEEE Transactions on Automatic control*, 57(3):592–606, 2012.

[5] R. Durrett. *Random graph dynamics*, volume 200. Cambridge university press Cambridge, 2007.

[6] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[7] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[8] I. Lobel and A. Ozdaglar. Distributed subgradient methods for convex optimization over random networks. *IEEE Transactions on Automatic Control*, 56(6):1291–1306, 2011.

[9] A. Makhdoumi and A. Ozdaglar. Convergence rate of distributed admm over networks. *IEEE Transactions on Automatic Control*, 62(10):5082–5095, 2017.

[10] A. Mokhtari, Q. Ling, and A. Ribeiro. Network newton distributed optimization methods. *IEEE Transactions on Signal Processing*, 65(1):146–161, 2017.

[11] A. Nedić. Asynchronous broadcast-based convex optimization over a network. *IEEE Transactions on Automatic Control*, 56(6):1337–1351, 2011.

[12] A. Nedić, A. Olshevsky, and M. G. Rabbat. Network topology and communication-computation tradeoffs in decentralized optimization. *arXiv preprint arXiv:1709.08765*, 2017.

[13] A. Nedić, A. Olshevsky, and W. Shi. Achieving geometric convergence for distributed optimization over time-varying graphs. *SIAM Journal on Optimization*, 27(4):2597–2633, 2017.

[14] A. Nedić and A. Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1):48–61, 2009.

[15] G. Qu and N. Li. Accelerated distributed nesterov gradient descent. *arXiv preprint arXiv:1705.07176*, 2017.

[16] G. Qu and N. Li. Harnessing smoothness to accelerate distributed optimization. *IEEE Transactions on Control of Network Systems*, 2017.

[17] K. Scaman, F. Bach, S. Bubeck, Y. T. Lee, and L. Massoulié. Optimal algorithms for smooth and strongly convex distributed optimization in networks. *arXiv preprint arXiv:1702.08704*, 2017.

[18] W. Shi, Q. Ling, K. Yuan, G. Wu, and W. Yin. On the linear convergence of the admm in decentralized consensus optimization. *IEEE Trans. Signal Processing*, 62(7):1750–1761, 2014.

[19] Z. Shi and A. Eryilmaz. Technical report. http://www2.ece.ohio-state.edu/~eryilmaz/papers/FlexibleDistOpt_Inf19.pdf, 2019.

[20] D. Tse and P. Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[21] K. I. Tsianos, S. Lawlor, and M. G. Rabbat. Push-sum distributed dual averaging for convex optimization. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 5453–5458. IEEE, 2012.

[22] J. Tsitsiklis, D. Bertsekas, and M. Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on automatic control*, 31(9):803–812, 1986.

[23] R. Tutunov, H. B. Ammar, and A. Jadbabaie. A distributed newton method for large scale consensus optimization. *arXiv preprint arXiv:1606.06593*, 2016.