Communication-Efficient Distributed Asynchronous ADMM

Sagar Shrestha
Department of Electrical Engineering and Computer Science
Oregon State University
shressag@oregonstate.edu

Abstract

In distributed optimization and federated learning, asynchronous *alternating direction method of multipliers* (ADMM) serves as an attractive option for large-scale optimization, data privacy, straggler nodes and variety of objective functions. However, communication costs can become a major bottleneck when the nodes have limited communication budgets or when the data to be communicated is prohibitively large. In this project, we propose introducing coarse quantization to the data to be exchanged in aynchronous ADMM so as to reduce communication overhead for large-scale federated learning and distributed optimization applications. We experimentally verify the convergence of the proposed method for several distributed learning tasks, including neural networks.

1 Introduction

In federated learning, numerous connected devices seek to learn a common statistical model using but without sharing their private data. It naturally demands for distributed large-scale optimization methods. Therefore, ADMM is a attractive choice because of its ability to handle large classes of optimization problems including non-smooth objective [1]. However, the heterogeneous nature of the network encountered in federated learning makes it inefficient for synchronous updates as it does not fully utilize the node's computation and communication capabilities [2]. As such, asynchronous ADMM has been proposed as a suitable method in [3], [4].

While asynchronous ADMM appears to fix the issue of stragglers, a major hurdle in its application for large scale models is the communication bottleneck. When the data to be communicated is large, e.g., neural network parameters, or the communication resources are limited, e.g. battery operated devices, exacerbated by hundreds to millions of iterations needed for convergence, communication time can become a major bottleneck in the learning process. To this end, we propose to utilize coarse quantization of the data to be communicated for both exact and inexact asynchronous ADMM update. We show empirically that we are able to reduce around 90% communication overhead for both exact and inexact asynchronous ADMM without any apparent loss in the convergence properties of the unquantized version.

2 Related Works

Asynchronous ADMM for inexact primal updates was proposed in [5] and for exact updates in [3,4]. However, communication cost reduction was not considered in those references. Recently, [6–8] considered inexact ADMM for the synchronous case in the context of federated learning. Although [8] claim communication efficiency of their method, the gain is only derived from reduction

in communication round from multiple local updates. However, communication requirement in each outer iteration can still be prohibitive for large scale problems. In addition to this effort, the proposed method in this project significantly reduces the communication overhead of each round. Further, the aforementioned references only provide simulations for convex problems (e.g., linear regression, logistic regression, LASSO, etc). In contrast, we provide simulations validating the proposed method in the case of deep neural networks, which is more representative of federated learning applications.

Closely related to ADMM, [9] proposed a primal-dual method utilizing compression and error-feedback to achieve communication efficiency in distributed optimization. However, their method is limited to the synchronous case and merely deals with the uplink communication overhead. In contrast, the proposed method in this projects reduces both the uplink and downlink communication overhead and incorporates the asynchronous update case, which is essential to dealing with stragglers in hetergeneous networks.

3 Background

In this section, we present the distributed asynchronous ADMM algorithm to provide necessary background for the proposed communication-efficient method.

3.1 ADMM

We are interested in solving the following optimization problem:

$$\underset{\boldsymbol{x}\in\mathbb{R}^M}{\text{minimize}}\; f(\boldsymbol{x}) + h(\boldsymbol{x}), \tag{1}$$

where $f : \mathbb{R}^M \to \mathbb{R}$ is a smooth cost function, and $h : \mathbb{R}^M \to \mathbb{R}$ is a convex (possibly non-smooth) regularization term. To solve the problem using N agents in a distributed setting, we require that $f(\boldsymbol{x})$ be decomposed into N local objectives:

$$f(\boldsymbol{x}) = \sum_{i=1}^{N} f_i(\boldsymbol{x}),$$

where $f_i : \mathbb{R}^M \to \mathbb{R}$ is a local objective function which only depends upon the data at node i. This allows us to reformulate problem (1) as a global variable consensus optimization problem as follows:

$$\begin{aligned} &\underset{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N,\boldsymbol{z}}{\text{minimize}} \sum_{i=1}^{N} f_i(\boldsymbol{x}_i) + h(\boldsymbol{z}) \\ &\text{subject to } \boldsymbol{x}_i = \boldsymbol{z},\; i = 1, 2, \ldots, N, \end{aligned} \tag{2}$$

where $\boldsymbol{z}$ is the consensus variable, and $\boldsymbol{x}_i$ is the node i's proxy for the global variable $\boldsymbol{z}$. Essentially, we wish to distribute the optimization of $f(\cdot)$ to N nodes, where each node optimizes for its local objective $f_i(\cdot)$. This results in an iterative optimization procedure such that in each iteration the nodes solve their corresponding local sub-problems, and communicate the solution to a central controller or the server. The server collects the local variables, updates the consensus variable and distributes it to all the nodes.

ADMM combines the features of dual ascent and augmented Lagrangian method to solve problem (2). The augmented Lagrangian of (2) can be written as follows:

$$\mathcal{L}\left(\{\boldsymbol{x}_i\}_{i=1}^N, \boldsymbol{z}, \{\boldsymbol{\lambda}_i\}_{i=1}^N\right) = \sum_{i=1}^{N} f_i(\boldsymbol{x}_i) + h(\boldsymbol{z}) + \boldsymbol{\lambda}_i^\top(\boldsymbol{x}_i - \boldsymbol{z}) + \frac{\rho}{2}\sum_{i=1}^{N}\|\boldsymbol{x}_i - \boldsymbol{z}\|_2^2, \tag{3}$$

where $\boldsymbol{\lambda}_i \in \mathbb{R}^M$ is the dual variable associated with the ith constraint and $\rho > 0$ is the penalty parameter. The above can be simplified as:

$$\mathcal{L}\left(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{u}\right) = \sum_{i=1}^{N} f_i(\boldsymbol{x}_i) + h(\boldsymbol{z}) + \frac{\rho}{2}\sum_{i=1}^{N}\|\boldsymbol{x}_i - \boldsymbol{z} + \boldsymbol{u}_i\|_2^2, \tag{4}$$

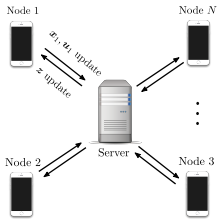

Figure 1: Illustration of Distributed Optimization using ADMM.

where $\boldsymbol{x} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]^\top$, $\boldsymbol{u} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_N]^\top$, and $\boldsymbol{u}_i = \boldsymbol{\lambda}_i/\rho$. ADMM involves iteratively optimizing all the primal variables, $\{\boldsymbol{x}_i\}_{i=1}^N$ and $\boldsymbol{z}$, followed by one step gradient ascent of the dual variables in a Gauss-Seidel fashion [1]. Figure 1 illustrates the distributed ADMM scenario. The nodes and the server are connected in a star topology and communicate local and consensus variables with each other. Following steps summarize one step of ADMM update:

$$\boldsymbol{x}_i^{(r+1)} \leftarrow \underset{\boldsymbol{x}_i \in \mathbb{R}^M}{\arg\min}\ f_i(\boldsymbol{x}_i) + \frac{\rho}{2}\|\boldsymbol{x}_i - \boldsymbol{z}^{(r)} + \boldsymbol{u}_i^{(r)}\|_2^2,\ i = 1, \ldots, N \tag{5a}$$

$$\boldsymbol{z}^{(r+1)} \leftarrow \underset{\boldsymbol{z} \in \mathbb{R}^M}{\arg\min}\ h(\boldsymbol{z}) + \frac{\rho}{2}\sum_{i=1}^{N}\|\boldsymbol{x}_i^{(r+1)} - \boldsymbol{z} + \boldsymbol{u}_i^{(r)}\|_2^2 \tag{5b}$$

$$\boldsymbol{u}_i^{(r+1)} \leftarrow \boldsymbol{u}_i^{(r)} + (\boldsymbol{x}_i^{(r+1)} - \boldsymbol{z}^{(r+1)}),\ i = 1, \ldots, N, \tag{5c}$$

where r is the current iteration index. Note that the updates (5a) and (5c) can be performed locally at the nodes and only requires the consensus variable. (5b), on the other hand, requires all local variables: $\{\boldsymbol{x}_i\}_{i=1}^N, \{\boldsymbol{u}_i\}_{i=1}^N$, and is carried out at the server. Therefore, the updates at node i are as follows:

$$\boldsymbol{x}_i^{(r+1)} \leftarrow \underset{\boldsymbol{x}_i \in \mathbb{R}^M}{\arg\min}\ f_i(\boldsymbol{x}_i) + \frac{\rho}{2}\|\boldsymbol{x}_i - \boldsymbol{z}^{(r)} + \boldsymbol{u}_i^{(r)}\|_2^2,\ i = 1, \ldots, N \tag{6a}$$

$$\boldsymbol{u}_i^{(r+1)} \leftarrow \boldsymbol{u}_i^{(r)} + (\boldsymbol{x}_i^{(r+1)} - \boldsymbol{z}^{(r)}),\ i = 1, \ldots, N. \tag{6b}$$

Send $\boldsymbol{x}_i^{(r+1)}, \boldsymbol{u}_i^{(r+1)}$ to the server.

Similarly the server update is as follows:

$$\boldsymbol{z}^{(r+1)} \leftarrow \underset{\boldsymbol{z} \in \mathbb{R}^M}{\arg\min}\ h(\boldsymbol{z}) + \frac{\rho}{2}\sum_{i=1}^{N}\|\boldsymbol{x}_i^{(r+1)} - \boldsymbol{z} + \boldsymbol{u}_i^{(r+1)}\|_2^2. \tag{7a}$$

Broadcast $\boldsymbol{z}^{(r+1)}$ to the nodes.

3.2 Asynchronous ADMM

The distributed implementation in (6) and (7) can be viewed as synchronous ADMM because all the nodes are synchronized, i.e., all nodes complete one update before the server makes its update. Therefore this distributed implementation retains the same convergence properties of the undistributed version. However, the speed of such method is limited to the speed of the slowest node. This does not fully utilize the computation abilities of the faster nodes (one with more computation and/or communication resources) in the network.

To address the limitation of synchronous setting, asynchronous ADMM has been considered for exact updates of primal variables in [3, 4] and inexact updates in [5]. In the asynchronous setting, the server does not wait for all the nodes to complete their updates. Instead the server performs its computation using updates from a subset of nodes. Specifically, let P be the minimum number of nodes that can trigger a server update. Then during iteration r, let $\mathcal{A}_r \subseteq \mathcal{V} = \{1, \ldots, N\}$ be the set of nodes that have completed its operation. The server waits until $|\mathcal{A}_r| \geq P$ before performing the

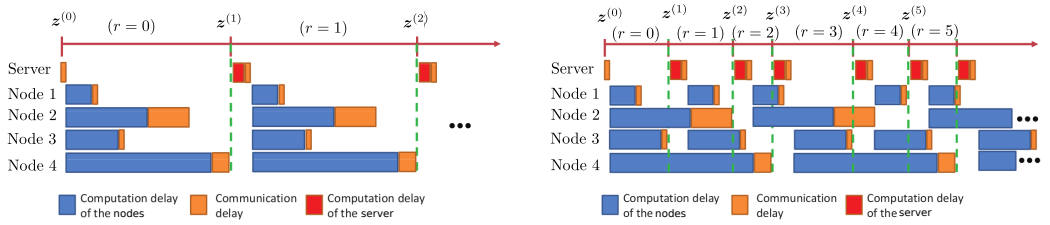

Figure 2: Illustration of synchronous and asynchronous distributed ADMM updates for $P = 2$. (Figure reproduced from [3]).

server update. In addition, in order to ensure that updates from all nodes arrive within a limited time frame, we define τ as the maximum delay (in iterations) allowed for the arrival of any node. This means that in any iteration, the server waits for the nodes that have not updated for $\tau - 1$ iterations.

With this, we can write the the node update as follows:

$$\boldsymbol{x}_i^{(r+1)} \leftarrow \begin{cases} \arg\min_{\boldsymbol{x}_i} \; f_i(\boldsymbol{x}_i) + \frac{\rho}{2}\|\boldsymbol{x}_i - \boldsymbol{z}^{(r)} + \boldsymbol{u}_i^{(r)}\|_2^2, & \forall i \in \mathcal{A}_r \\ \boldsymbol{x}_i^{(r)} \quad \forall i \in \mathcal{V}\backslash\mathcal{A}_r \end{cases} \tag{8a}$$

$$\boldsymbol{u}_i^{(r+1)} \leftarrow \begin{cases} \boldsymbol{u}_i^{(r)} + (\boldsymbol{x}_i^{(r+1)} - \boldsymbol{z}^{(r)}), & \forall i \in \mathcal{A}_r \\ \boldsymbol{u}_i^{(r)}, \quad \forall i \in \mathcal{V}\backslash\mathcal{A}_r \end{cases} \tag{8b}$$

Send $\boldsymbol{x}_i^{(r+1)}, \boldsymbol{u}_i^{(r+1)}$.

The server update is the same as (7).

Figure 2 illustrates the difference between the synchronous and asynchronous ADMM for $P = 2$. We can see that in the synchronous update the server waits for all the nodes to finish computation and communication before starting its update. Whereas, in the asynchronous update, the server starts its computation after receiving updates from P nodes.

4 Communication-Efficient Asynchronous ADMM

Asynchronous ADMM allows utilization of computation and/or communication resources at the nodes. However, when M is large (e.g., in neural networks, M can be in the order of millions), the communication cost at each round can be prohibitive, not to mention the number of iterations required for convergence of the algorithm. For instance, if $M = 10,000,000$, each node needs to upload 640MB of data at each iteration. This is not feasible if the node is a mobile device or the communication frequency is large. Therefore, it is well-motivated to consider further communication reduction per each update. To this end, we propose a carefully designed communication reduction technique for asynchronous distributed ADMM.

4.1 Compression and Error-Feedback

We utilize the ideas from gradient compression literature [10–12] to reduce the communication cost of the exchanged information. Specifically, we introduce a compression operator, $\boldsymbol{\mathcal{C}} : \mathbb{R}^M \to \mathcal{Q}^M$. Here $\mathcal{Q}$ is the quantized domain, a subset of real valued domain. The idea is that

$$\boldsymbol{\mathcal{C}}(\boldsymbol{y}) \approx \boldsymbol{y},$$

but $\boldsymbol{\mathcal{C}}(\boldsymbol{y})$ requires much fewer number of bits to represent than $\boldsymbol{y}$. The compressor can be quantization based [11,13] or sparsification based [10,14]. Moreover, we note that instead of communicating $\boldsymbol{\mathcal{C}}(\boldsymbol{y}^{(r)})$ in iteration r, we can reduce the compression error by communicating $\boldsymbol{\mathcal{C}}(\boldsymbol{y}^{(r+1)} - \boldsymbol{y}^{(r)})$. This is because the change in the iterate, $\boldsymbol{y}^{(r+1)} - \boldsymbol{y}^{(r)}$, is supposed to converge to zero for any converging algorithm.

In addition to compressing the information to be exchanged, we utilize error-feedback method introduced in [12] that has been shown to be important to ensure convergence in some compressor (e.g., [11]) or improve convergence rate in others (e.g., [13]). To explain error-feedback, let $\boldsymbol{y}^{(r)}$ be the iterate we want to communicate from a source to a destination. Let the iterate be obtained at the source by the following relation:

$$\boldsymbol{y}^{(r+1)} \leftarrow \boldsymbol{y}^{(r)} + \boldsymbol{g}^{(r)},$$

where $\boldsymbol{g}^{(r)}$ is the change in the iterate in iteration r. Let $\widehat{\boldsymbol{y}}^{(r)}$ be the estimate of $\boldsymbol{y}^{(r)}$ at the destination, which is obtained as follows:

$$\begin{aligned}
\widehat{\boldsymbol{y}}^{(r+1)} &\leftarrow \widehat{\boldsymbol{y}}^{(r)} + \boldsymbol{\mathcal{C}}(\boldsymbol{g}^{(r)}) \\
&= \widehat{\boldsymbol{y}}^{(r)} + \boldsymbol{g}^{(r)} + \boldsymbol{\delta}^{(r)} \\
&= \boldsymbol{y}^{(0)} + \sum_{t=0}^{r} \boldsymbol{g}^{(t)} + \sum_{t=0}^{r} \boldsymbol{\delta}^{(t)} \\
&= \boldsymbol{y}^{(r+1)} + \underbrace{\sum_{t=0}^{r} \boldsymbol{\delta}^{(t)}}_{\text{aggregated error}},
\end{aligned}$$

where $\boldsymbol{\delta}^{(r)}$ is the compression error in iteration r. We can see that the error in each iteration keeps integrating. Therefore when r becomes large, $\widehat{\boldsymbol{y}}^{(r+1)}$ can be very far from $\boldsymbol{y}^{(r)}$. In error-feedback, we *feedback* the error of previous iteration, $\boldsymbol{\delta}^{(r-1)}$, along with the $\boldsymbol{g}^{(r)}$. Therefore, the source transmits $\boldsymbol{\mathcal{C}}(\boldsymbol{g}^{(r)} - \boldsymbol{\delta}^{(r-1)})$. This results in the following update of the estimate at the destination:

$$\begin{aligned}
\widehat{\boldsymbol{y}}^{(r+1)} &\leftarrow \widehat{\boldsymbol{y}}^{(r)} + \boldsymbol{\mathcal{C}}(\boldsymbol{g}^{(r)} - \boldsymbol{\delta}^{(r-1)}) \\
&= \widehat{\boldsymbol{y}}^{(r)} + \boldsymbol{g}^{(r)} - \boldsymbol{\delta}^{(r-1)} + \boldsymbol{\delta}^{(r)} \\
&= \boldsymbol{y}^{(0)} + \sum_{t=0}^{r} \boldsymbol{g}^{(t)} + \sum_{t=0}^{r} (\boldsymbol{\delta}^{(t)} - \boldsymbol{\delta}^{(t-1)}) \\
&= \boldsymbol{y}^{(r+1)} + \boldsymbol{\delta}^{(r)}.
\end{aligned}$$

This shows that we have eliminated the error terms from previous iterations ensuring that $\widehat{\boldsymbol{y}}^{(r)}$ is close to $\boldsymbol{y}^{(r)}$ given sufficiently accurate compressor.

4.2 Asynchronous ADMM with Compression and Error-Feedback

Now, let us apply compression along with error feedback to asynchronous ADMM. Let $\widehat{\boldsymbol{x}}_i^{(r)}$ and $\widehat{\boldsymbol{u}}_i^{(r)}$ be the server's estimates of $\boldsymbol{x}_i^{(r)}$ and $\boldsymbol{u}_i^{(r)}$ in iteration r. Similarly, let $\widehat{\boldsymbol{z}}^{(r)}$ be the nodes' estimate of $\boldsymbol{z}^{(r)}$. Then the node and server operations are carried out as follows:

Node Operations. In iteration r, node i updates its local variables using its estimate of the $\boldsymbol{z}^{(r)}$, given by $\widehat{\boldsymbol{z}}^{(r)}$, as follows:

$$\boldsymbol{x}_i^{(r+1)} \leftarrow \begin{cases} \arg\min_{\boldsymbol{x}_i} \; f_i(\boldsymbol{x}_i) + \frac{\rho}{2}\|\boldsymbol{x}_i - \widehat{\boldsymbol{z}}^{(r)} + \boldsymbol{u}_i^{(r)}\|_2^2, & \text{if } i \in \mathcal{A}_r \\ \boldsymbol{x}_i^{(r)} \quad \text{if } i \in \mathcal{V}\backslash\mathcal{A}_r \end{cases} \tag{9a}$$

$$\boldsymbol{u}_i^{(r+1)} \leftarrow \begin{cases} \boldsymbol{u}_i^{(r)} + (\boldsymbol{x}_i^{(r+1)} - \widehat{\boldsymbol{z}}^{(r)}), & \text{if } i \in \mathcal{A}_r \\ \boldsymbol{u}_i^{(r)}, \quad \text{if } i \in \mathcal{V}\backslash\mathcal{A}_r \end{cases} \tag{9b}$$

Node i computes the data to be communicated to the server, $\boldsymbol{\Delta}_{\boldsymbol{x}_i}^{(r)}$ and $\boldsymbol{\Delta}_{\boldsymbol{u}_i}^{(r)}$, change of the iterates along with the error from previous iteration as follows:

$$\boldsymbol{\Delta}_{\boldsymbol{x}_i}^{(r)} = \underbrace{\boldsymbol{x}_i^{(r+1)} - \boldsymbol{x}_i^{(r)}}_{\text{current change}} + \underbrace{\boldsymbol{x}_i^{(r)} - \widehat{\boldsymbol{x}}_i^{(r)}}_{\text{previous error}} = \boldsymbol{x}_i^{(r+1)} - \widehat{\boldsymbol{x}}_i^{(r)} \tag{10}$$

$$\boldsymbol{\Delta}_{\boldsymbol{u}_i}^{(r)} = \boldsymbol{u}_i^{(r+1)} - \widehat{\boldsymbol{u}}_i^{(r)} \tag{11}$$

$$\tag{12}$$

Node i then sends $\mathcal{C}(\boldsymbol{\Delta}_{\boldsymbol{x}_i}^{(r)})$ and $\mathcal{C}(\boldsymbol{\Delta}_{\boldsymbol{u}_i}^{(r)})$ to the server. The server then updates its estimate of node i's local variables as follows:

$$\widehat{\boldsymbol{x}}_i^{(r+1)} \leftarrow \widehat{\boldsymbol{x}}_i^{(r)} + \mathcal{C}(\boldsymbol{\Delta}_{\boldsymbol{x}_i}^{(r)}), \tag{13}$$

$$\widehat{\boldsymbol{u}}_i^{(r+1)} \leftarrow \widehat{\boldsymbol{u}}_i^{(r)} + \mathcal{C}(\boldsymbol{\Delta}_{\boldsymbol{u}_i}^{(r)}). \tag{14}$$

Note that above operation is also carried out at the nodes as they requires $\widehat{\boldsymbol{x}}_i^{(r+1)}$ to compute the error term (for error feedback) in the next iteration.

Server Operations. The server updates $\boldsymbol{z}$ with the estimates of the node variables as follows:

$$\boldsymbol{z}^{(r+1)} \leftarrow \underset{\boldsymbol{z}\in\mathbb{R}^M}{\arg\min}\ h(\boldsymbol{z}) + \frac{\rho}{2}\sum_{i=1}^{N} \|\widehat{\boldsymbol{x}}_i^{(r+1)} - \boldsymbol{z} + \widehat{\boldsymbol{u}}_i^{(r+1)}\|_2^2.. \tag{15}$$

Similar to the operation in the nodes, the server computes $\boldsymbol{\Delta}_{\boldsymbol{z}}^{(r)}$ as $\boldsymbol{\Delta}_{\boldsymbol{z}}^{(r)} = \boldsymbol{z}^{(r+1)} - \widehat{\boldsymbol{z}}^{(r)}$. The server then broadcasts $\mathcal{C}(\boldsymbol{\Delta}_{\boldsymbol{z}}^{(r)})$ to all the nodes. The nodes and the server update the node's estimate of $\boldsymbol{z}$ as follows:

$$\widehat{\boldsymbol{z}}^{(r+1)} \leftarrow \widehat{\boldsymbol{z}}^{(r)} + \mathcal{C}(\boldsymbol{\Delta}_{\boldsymbol{z}}^{(r)}). \tag{16}$$

The algorithm is referred to as *Quantized* ADMM (QADMM) and summarized in Algorithm 1. Note that we use a subroutine, `simulate-async()`, in order to simulate the asynchronous scenario. Specifically, we assume that `simulate-async()` is an oracle that provides us the set of nodes that will complete their computation and communication within the next iteration.

Choice of Compressor. In the simulations, we use a random compressor introduced in [13], which allows for multi-precision quantization and has favourable convergence properties in practice.

To quantize $\boldsymbol{\Delta} \in \mathbb{R}^M$, $\boldsymbol{\Delta} \neq \boldsymbol{0}$, $\mathcal{C}(\boldsymbol{\Delta})$ is computed as follows. We first divide the range from 0 to 1 into S intervals of equal width, where S is related to the number of levels of quantization. Specifically, if q is the number of bits used to represent a scalar value, $S = 2^{q-1} - 1$. The quantization operation operates elementwise. Therefore, for each element $\boldsymbol{\Delta}(m)$, we can find an interval $[p/S, (p+1)/S], p \in \{0, \ldots, S-1\}$, such that the normalized value $|\boldsymbol{\Delta}(j,k)|/\|\boldsymbol{\Delta}\|_{\max} \in [p/S, (p+1)/S]$. Next, a Bernoulli random variable, $h(\boldsymbol{\Delta}(j,k), S)$, is defined as follows:

$$h(\boldsymbol{\Delta}(j,k), S) = \begin{cases} p/S & \text{w.p.} \quad 1 - \left(\frac{|\boldsymbol{\Delta}(j,k)|}{\|\boldsymbol{\Delta}\|_{\max}} S - p\right) \\ (p+1)/S & \text{otherwise.} \end{cases} \tag{17}$$

Finally, we unnormalize $h(\boldsymbol{\Delta}(j,k), S)$ using the sign and magnitude information as $[\mathcal{C}(\boldsymbol{\Delta})]_m = \|\boldsymbol{\Delta}\|_{\max}\mathrm{sgn}(\boldsymbol{\Delta}(m)) \cdot h(\boldsymbol{\Delta}(m), S)$, where $\mathrm{sgn}(\cdot)$ is the sign operator.

5 Simulation Results

In this section, We present simulations to validate the effectiveness of the proposed method.

5.1 LASSO

Consider the LASSO problem

$$\underset{\boldsymbol{x}\in\mathbb{R}^M}{\text{minimize}} \sum_{i=1}^{N} \|\boldsymbol{A}_i\boldsymbol{x} - \boldsymbol{b}_i\|_2^2 + \theta\|\boldsymbol{x}\|_1, \tag{18}$$

where $\boldsymbol{A}_i \in \mathbb{R}^{H\times M}$, $\boldsymbol{b}_i \in \mathbb{R}^H$, $\theta > 0$.

We reformulate (18) to fit ADMM based distributed optimizatiion framework as follows:

$$\begin{aligned} &\underset{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N,\boldsymbol{z}}{\text{minimize}} \ \sum_{i=1}^{N} \|\boldsymbol{A}_i\boldsymbol{x}_i - \boldsymbol{b}_i\|_2^2 + \theta\|\boldsymbol{z}\|_1 \\ &\text{subject to} \ \ \boldsymbol{x}_i = \boldsymbol{z} \quad i = 1, \ldots, N. \end{aligned}$$

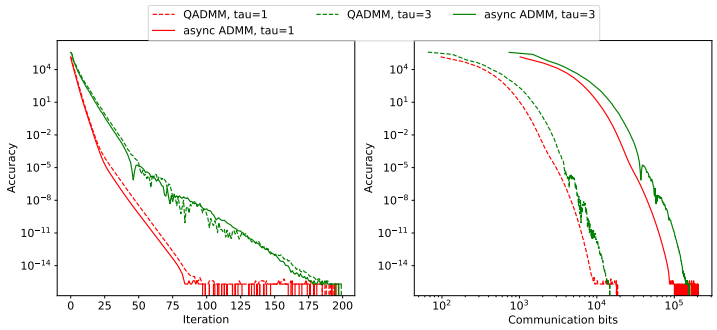

Figure 3: Classification accuracy attained by the proposed method, QADMM, with respect to the unquantized version for various τ vs. iterations and communication bits.

We can see that $f_i(\boldsymbol{x}_i) = \|\boldsymbol{A}_i\boldsymbol{x}_i - \boldsymbol{b}_i\|_2^2$. Therefore, each node operation only depends upon its local data $\boldsymbol{A}_i$ and $\boldsymbol{b}_i$. The primal update in (9a) is a least square problem which is solved optimally at each iteration. Similarly the consensus update in (15) is a proximal update with respect to L-1 norm. The optimal solution is given by soft-thresholding operation. Therefore, this is an example of exact minimization based QADMM.

We generate the elements of $\boldsymbol{A}_i$ using standard normal distribution, i.e., $\sim \mathcal{N}(0,1)$.$\boldsymbol{b}_i$'s are generated following $\boldsymbol{b}_i = \boldsymbol{A}_i\boldsymbol{z}_0 + \boldsymbol{n}_i$, where $\boldsymbol{z}_0 \in \mathbb{R}^M$ is a sparse random vector with $0.2M$ non-zero elements sampled from $\mathcal{N}(0,1)$, and $\boldsymbol{n}_i$ is noise vector sampled from $\mathcal{N}(0, 0.01)$. We use $N = 16$. To simulate the asynchronous case, we implement `simulate-async()` subroutine in Algorithm 1 by the following procedure. We randomly split N nodes into two sets. For each element of the first set, we set the probability of getting selected as 0.1 and for the other half, the probability is set to 0.8. Thus the nodes selected by `simulate-async()` complete their local updates and communicate them to the server within the next iteration.

Metric. To measure the progress of the algorithm towards the optimal, we observe the accuracy defined as follows:

$$\text{Accuracy}(r) = \frac{|\mathcal{L}(\boldsymbol{x}^{(r)}, \boldsymbol{z}^{(r)}, \boldsymbol{u}^{(r)}) - F^\star|}{F^\star} \tag{19}$$

where $\mathcal{L}(\cdot)$ is the augmented lagrangian in (4), and $F^\star$ is the optimal objective value for (18) In order to evaluate the gain in communication efficiency, we define communication bits as follows:

$$\text{Communication bits} = \frac{\text{total bits communicated between the nodes and the server}}{M} \tag{20}$$

Baseline. We compare QADMM with its unquantized version referred to as `async ADMM`.

Figure 3 shows the accuracy with respect to iteration and communication bits for the case when $(M, \rho, \theta, N, H) = (200, 500, 0.1, 16, 100)$ for $\tau \in \{1, 3\}$. The result is averaged over 10 Monte Carlo trials. Note that $\tau = 1$ corresponds to the synchronous case. We use $q = 3$ bits per scalar for both the downlink and uplink compression. We can see that there is apparently no degradation in convergence behavior due to compression. Further, the plot with respect to the communication bits shows that QADMM approaches the optimal much faster ($\approx 10\times$) than its unquantized version. Specifically, we observe that to obtain an accuracy of 10^{-10}, QADMM requires 90.62% less communication bits than the unquantized version.

5.2 MNIST Classifier

In this subsection, we validate the effectiveness of the proposed method for neural-network based classifiers. Since many of the recent applications of federated learning employ neural network based models, this example serves to validate the usefulness of QADMM in real-world applications.

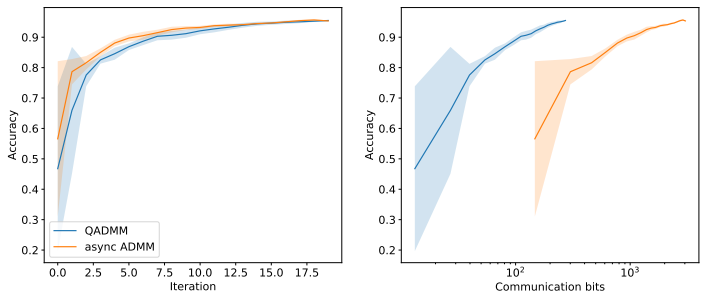

Figure 4: Classification accuracy attained by the proposed method, `QADMM`, with respect to the unquantized version vs. iterations and communication bits.

We consider a *convolutional neural network* (CNN) based classifier with 6 layers, where the first 5 layers are convolutional layers and the final layer is fully connected. The filter size for the first 5 layers is 3×3, stride of 2, padding of 1, and number of filters are 16, 32, 64, 128, and 128 respectively. The final fully connected layer has 10 neurons to classify among the 10 digits with a sigmoid activation at the output. The total number of parameters $M = 246,762$. Note that we cannot exactly solve the primal update in (9a) in this case because of the highly non-convex nature of the problem. Therefore, we return inexact solution by running 10 iterations of gradient descent with a batch size of 64 at each update. We use ADAM [15] with an initial learning rate of 0.001 for the inexact primal update. Consequently, this is an example of inexact asynchronous ADMM.

We use $N = 3$, and randomly divide the $60,000$ training examples into N partitions. Similar to the previous example, for each call to the subroutine `simulate-async`(), we create two groups of nodes with each node independently assigned to either of the groups with equal probability. The nodes in the first group have 0.1 probability of being selected by the subroutine and those in the second group have 0.8 probability of being selected. We set the number of bits $q = 3$ and $\tau = 3$.

Figure 4 shows the classification accuracy attained by `QADMM` and `async ADMM` with respect to iterations and communication cost on a held out test set of size 10,000. The result is averaged over 5 Monte Carlo trials. We can see that there is virtually no degradation in convergence properties of the proposed method relative to the unquantized version. However, there is a significant reduction in communication bits required to reach the desired classification accuracy. Specifically, to attain a classification accuracy of 95%, `QADMM` requires 91.02% less communication bits than the unquantized version.

6 Conclusion

In this project, we proposed a communication-efficient method for asynchronous distributed implementation of exact and inexact ADMM. With carefully designed compression scheme and error feedback, we showed experimentally that the proposed method can reduce communication cost by more than 90% in both uplink and downlink direction for both convex problems as well as non-convex problem involving deep neural networks.

References

[1] S. Boyd, N. Parikh, and E. Chu, *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011.

[2] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[3] T.-H. Chang, M. Hong, W.-C. Liao, and X. Wang, "Asynchronous distributed admm for large-scale optimization—part i: Algorithm and convergence analysis," *IEEE Transactions on Signal Processing*, vol. 64, no. 12, pp. 3118–3130, 2016.

[4] R. Zhang and J. Kwok, "Asynchronous distributed admm for consensus optimization," in *International conference on machine learning*. PMLR, 2014, pp. 1701–1709.

[5] M. Hong, "A distributed, asynchronous and incremental algorithm for nonconvex optimization: An admm based approach," *arXiv preprint arXiv:1412.6058*, 2014.

[6] M. Ryu and K. Kim, "Differentially private federated learning via inexact admm," *arXiv preprint arXiv:2106.06127*, 2021.

[7] S. Yue, J. Ren, J. Xin, S. Lin, and J. Zhang, "Inexact-admm based federated meta-learning for fast and continual edge learning," in *Proceedings of the Twenty-second International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, 2021, pp. 91–100.

[8] S. Zhou and G. Y. Li, "Communication-efficient admm-based federated learning," *arXiv preprint arXiv:2110.15318*, 2021.

[9] C. Chen, J. Zhang, L. Shen, P. Zhao, and Z. Luo, "Communication efficient primal-dual algorithm for nonconvex nonsmooth distributed optimization," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2021, pp. 1594–1602.

[10] D. Basu, D. Data, C. Karakus, and S. Diggavi, "Qsparse-local-SGD: Distributed SGD with quantization, sparsification and local computations," in *Proc. NeurIPS*, vol. 32, 2019.

[11] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signSGD: Compressed optimisation for non-convex problems," in *Proc. ICML*. PMLR, 2018, pp. 560–569.

[12] S. P. Karimireddy, Q. Rebjock, S. Stich, and M. Jaggi, "Error feedback fixes signSGD and other gradient compression schemes," in *Proc. ICML*. PMLR, 2019, pp. 3252–3261.

[13] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. NeurIPS*, vol. 30, 2017, pp. 1709–1720.

[14] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified SGD with memory," in *Proc. NeurIPS*, vol. 31, 2018, pp. 4447–4458.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

Algorithm 1: `QADMM`

`// Initialization at the nodes`

for $i \leftarrow 1 : N$ **do**
  Initialize $\boldsymbol{x}_i^{(0)}, \boldsymbol{u}_i^{(0)}$;
  Transmit $\boldsymbol{x}_i^{(0)}, \boldsymbol{u}_i^{(0)}$ to the server using full precision (e.g., 32-bits per scalar);
end

`// Initialization at the server`

$d_1 = \cdots = d_N = 0$;
$\widehat{\boldsymbol{x}}_i^{(0)} \leftarrow \boldsymbol{x}_i^{(0)}, \quad \widehat{\boldsymbol{u}}_i^{(0)} \leftarrow \boldsymbol{u}_i^{(0)}, \forall i;$ `// received from the nodes with full precision`
$\boldsymbol{z}^{(0)} \leftarrow \arg\min_{\boldsymbol{z}} \; h(\boldsymbol{z}) + \frac{\rho}{2} \sum_{i=1}^{N} \|\widehat{\boldsymbol{x}}_i^{(0)} - \boldsymbol{z} + \widehat{\boldsymbol{u}}_i^{(0)}\|_2^2$;
Broadcast $\boldsymbol{z}^{(0)}$ to the nodes using full precision;
$r \leftarrow 0$;
while *some stopping criteria is not met* **do**
  `// At the nodes`
  for $i = 1 : N$ *in parallel* **do**
    if *r = 0* **then**
      $\widehat{\boldsymbol{z}}^{(0)} \leftarrow \boldsymbol{z}^{(0)}$; `// received from the server with full precision`
    else
      Receiving $\boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{z}}^{(r-1)})$ from the server;
      $\widehat{\boldsymbol{z}}^{(r)} \leftarrow \widehat{\boldsymbol{z}}^{(r-1)} + \boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{z}}^{(r-1)})$;
    end
    if *node* $i \in \mathcal{A}_r$ **then**
      $\boldsymbol{x}_i^{(r+1)} \leftarrow \arg\min_{\boldsymbol{x}_i} \; f_i(\boldsymbol{x}_i) + \frac{\rho}{2} \|\boldsymbol{x}_i - \widehat{\boldsymbol{z}}^{(r)} + \boldsymbol{u}_i^{(r)}\|_2^2$;
      $\boldsymbol{u}_i^{(r+1)} \leftarrow \boldsymbol{u}_i^{(r)} + (\boldsymbol{x}_i^{(r+1)} - \widehat{\boldsymbol{z}}^{(r)})$;
      Send $\{\boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{x}_i}^{(r)}), \boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{u}_i}^{(r)})\}$ to the server;
    else
      $\boldsymbol{x}_i^{(r+1)} \leftarrow \boldsymbol{x}_i^{(r)}$;
      $\boldsymbol{u}_i^{(r+1)} \leftarrow \boldsymbol{u}_i^{(r)}$;
    end
  end
  `// At the Server`
  Receive $\{\boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{x}_i}^{(r)}), \boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{u}_i}^{(r)})\}_{i \in \mathcal{A}_r}$ from the nodes such that $|\mathcal{A}_r| \geq P$
  $\mathcal{A}_{r+1} \leftarrow$ `simulate-async()`; **for** $i \in \mathcal{A}_r$ **do**
    $\widehat{\boldsymbol{x}}_i^{(r+1)} \leftarrow \widehat{\boldsymbol{x}}_i^{(r)} + \boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{x}_i}^{(r)})$;
    $\widehat{\boldsymbol{u}}_i^{(r+1)} \leftarrow \widehat{\boldsymbol{u}}_i^{(r)} + \boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{u}_i}^{(r)})$;
    $d_i \leftarrow 0$;
  end
  for $i \in \mathcal{V} \backslash \mathcal{A}_r$ **do**
    if $d_i = \tau - 1$ **then**
      $\mathcal{A}_{r+1} \leftarrow \mathcal{A}_{r+1} \cup \{i\}$;
    end
    $\widehat{\boldsymbol{x}}_i^{(r+1)} \leftarrow \widehat{\boldsymbol{x}}_i^{(r)}$;
    $\widehat{\boldsymbol{u}}_i^{(r+1)} \leftarrow \widehat{\boldsymbol{u}}_i^{(r)}$;
    $d_i \leftarrow d_i + 1$;
  end
  $\boldsymbol{z}^{(r+1)} \leftarrow \arg\min_{\boldsymbol{z}} \; h(\boldsymbol{z}) + \frac{\rho}{2} \sum_{i=1}^{N} \|\widehat{\boldsymbol{x}}_i^{(r+1)} - \boldsymbol{z} + \widehat{\boldsymbol{u}}_i^{(r+1)}\|_2^2$;
  Broadcast $\boldsymbol{\mathcal{C}}(\boldsymbol{\Delta}_{\boldsymbol{z}}^{(r)})$ to the nodes;
  $r \leftarrow r + 1$;
end
Output: $\boldsymbol{z}^{(r)}, \{\boldsymbol{x}_i^{(r)}\}_{i=1}^{N}$.