Compressed Coded Distributed Computing

Songze Li*, Mohammad Ali Maddah-Ali †, and A. Salman Avestimehr*
* Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA
† Nokia Bell Labs, Holmdel, NJ, USA

***Abstract*—Communication overhead is one of the major performance bottlenecks in large-scale distributed computing systems, especially for machine learning applications. Conventionally, compression techniques are used to reduce the load of communication by combining intermediate results of *the same* computation task as much as possible. Recently, via the development of coded distributed computing (CDC), it has been shown that it is possible to code across intermediate results of *different* tasks to further reduce communication. We propose a new scheme, named *compressed coded distributed computing* (in short, *compressed CDC*), which jointly exploits these two techniques (i.e., combining intermediate results of the same computation and coding across intermediate results of different computations) to significantly reduce the communication load for computations with linear aggregation of intermediate results in the final stage that are prevalent in machine learning (e.g., distributed training where partial gradients are computed distributedly and then averaged in the final stage). In particular, compressed CDC first compresses/combines several intermediate results for a single computation, and then utilizes multiple such combined packets to create a coded multicast packet that is simultaneously useful for multiple computations. We characterize the achievable communication load of compressed CDC and show that it substantially outperforms both combining methods and CDC scheme.**

I. Introduction

In order to scale up machine learning applications that process a massive amount of data, various distributed computing frameworks have been developed where data is stored and processed distributedly on multiple cores or GPUs on a single machine, or multiple machines in computing clusters (see, e.g., [1], [2]). When implementing these frameworks, the communication overhead of shuffling intermediate results across distributed computing nodes is a major performance bottleneck. For example, it was observed in [3] that on a Facebook's Hadoop cluster, 33% of the job execution time was spent on data shuffling. This bottleneck is becoming worse for training deep neural networks with millions of model parameters (e.g., ResNet-50 [4]) using distributed stochastic gradient descent, where partial gradients with millions of entries need to be passed between computing nodes.

Conventionally, compression techniques are used to reduce the communication load by combining intermediate results of *the same* computation task as much as possible. For example, in the original MapReduce framework [1], when the Reduce function is commutative and associative, a "combiner function" is proposed to pre-combine multiple intermediate values with the same key computed from different Map functions. Then, instead of sending multiple values to the reducer, the mapper sends the pre-combined value whose size is the same as one uncombined value, which significantly reduces the bandwidth consumption without any performance loss.

Coded distributed computing (CDC) is another approach that has been recently proposed in [5], [6] to mitigate the communication bottleneck. Unlike the compression/combining technique, CDC enables coding opportunities across intermediate results of *different* computation tasks to reduce the communication load. In particular, within a MapReduce-type model, CDC specifies a repetitive pattern of computing Map functions, creating side information at the computing nodes that enables coded multicasting during data shuffling across nodes, where each coded multicast packet is simultaneously useful for multiple Reduce tasks. For example, if we repeat each Map task r times across the cluster, then utilizing the CDC scheme, we can reduce the total amount of bandwidth consumption by r times. It has been shown that CDC can provide substantial speedups in practice [7], and several generalizations of it have been developed in the literature [8]–[11].

We focus on MapReduce-type distributed computing frameworks and propose a new scheme, named *compressed coded distributed computing* (in short, *compressed CDC*). It jointly exploits the above compression/combining technique and the CDC scheme to significantly reduce the communication load for computation tasks with linear Reduce functions that are prevalent in data analytics (e.g., distributed gradient descent where the partial gradients computed at multiple distributed computing nodes are averaged to reduce to the final gradient). In particular, the compressed CDC scheme specifies a repetitive storage of the dataset across distributed computing nodes. Each node, after processing locally stored files, first pre-combines several intermediate values of a single computation task needed by another node. Having generated multiple such pre-combined packets for different tasks, the node further codes them to generate a coded multicast packet that is simultaneously useful for multiple tasks. Therefore, compressed CDC enjoys both the intra-computation gain from combining, and the inter-computation gain from coded multicasting.

We characterize the achievable communication load of compressed CDC and show that it substantially outperforms both combining methods and CDC scheme. In particular, compared with the scheme that only relies on the combining technique, compressed CDC reduces the communication load by a factor that is proportional to the storage size of each computing node, which is significant for the common scenarios where large-scale machine learning tasks are executed on commodity servers with relatively small storage size. On the other hand, compared with the CDC scheme whose communication load scales linearly with the size of the dataset, compressed CDC eliminates this dependency by pre-combining intermediate

978-1-5386-4780-6/18/$31.00©2018 IEEE

values of the same task, allowing the system to scale up to handle computations on arbitrarily large dataset.

Other related work. Motivated by the fact that training algorithms exhibit tolerance to precision loss of intermediate results, as opposed to the above lossless compression technique that guarantees exact computation results, a family of lossy compression (or quantization) algorithms for distributed learning systems have been developed to compress the intermediate results (e.g., gradients) for a smaller bandwidth consumption (see, e.g., [12], [13]). Apart from compression, various coding techniques have also been recently utilized in distributed machine learning algorithms to mitigate the communication bottleneck and the straggler's delay (see, e.g. [14]–[22]).

II. Motivating Example

In this section, we demonstrate through a motivating example, how compression/combining and CDC techniques, applied alone or jointly, can help to reduce the bandwidth requirement for distributed computing tasks.

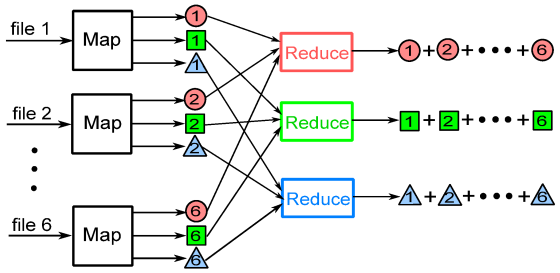

Fig. 1: A MapReduce framework to compute 3 functions from 6 files with linear Reduce functions.

As shown in Fig. 1, we consider a MapReduce job of computing 3 output functions, represented by red/circle, green/square, and blue/triangle respectively, by processing 6 input files. When mapping a file, we obtain 3 intermediate values, one for each of the functions, which are represented by the color/shape of the corresponding functions labelled by the file index. The Reduce operation of each output function computes its final result by summing up the intermediate values of the function from all 6 input files. This computation job is executed on 3 distributed computing nodes connected through a multicast network. Each node can store up to 4 files in its local memory. As shown in Fig. 2, we assign the computation tasks such that Nodes 1, 2, and 3 are respectively responsible for final reduction of red/circle, green/square, and blue/triangle functions. For this problem, we are interested in minimizing the communication load, which is the number of bits that need to be shuffled between computing nodes to accomplish the computation tasks, normalized by the size of a single intermediate value. Next, we describe three coded computing schemes, and compare their communication loads.

For all of these three schemes, as illustrated in Fig. 2, the file placement is performed such that Node 1 stores the files $1, 2, 3, 4$, Node 2 stores the files $3, 4, 5, 6$, and Node 3 stores the files $5, 6, 1, 2$.

1) Compression scheme: Since only the sum of the intermediate values is needed for final reduction, we can pre-combine the computed intermediate values of the same function at the sender node to reduce communication. For example, as shown in Fig. 2(a), having computed the green squares labelled by 1 and 2 in the Map phase, Node 1 sums them up and sends the computed sum to Node 2, instead of sending them individually. Upon receiving this pre-combined packet, Node 2 can directly use it for the final reduction of the green/square function. This compression scheme reduces the communication load by half, compared with the schemes that unicast uncombined intermediate values, and achieves a communication load of 3.

2) CDC scheme: Utilizing the redundant Map results across computing nodes, the CDC scheme creates coded multicast packets by combining intermediate values of different functions that are intended at different nodes. As shown in Fig. 2(b), since the blue triangle labelled by 3 is computed at both Nodes 1 and 2, and the green square labelled by 1 is computed at both Nodes 1 and 3, Node 1 can multicast the bit-wise XOR (denoted by $\oplus$) of these two intermediate values to the other two nodes. From this coded packet, both Nodes 2 and 3 can decode their intended values by cancelling their locally computed values. Since each of the multicast packets is simultaneously useful for two nodes, the CDC scheme cuts the communication load by half from the schemes that unicast uncoded intermediate values, and also achieves a communication load of 3. We note that since the CDC scheme allows to recover each of the intermediate values individually, it can be utilized on more general computations with arbitrary Reduce functions to slash the communication load.

3) Compressed CDC scheme: The above two schemes can be applied jointly to further reduce the communication load. Each node, as shown in Fig. 2(c), sums up two pairs of intermediate values to generate two pre-combined packets, each of which is needed by another node. Then, for example, Node 1 first splits each of its pre-combined packets (the unlabelled green square and the unlabelled blue triangle) into two segments, and computes the bitwise-XOR, of two segments, one from each of the pre-combined packets, generating a coded packet whose size is half of the size of an intermediate value. Finally, Node 1 multicasts this coded packet to Nodes 2 and 3. Similar operations are performed at Nodes 2 and 3. Next, each node utilizes its local computation results to decode the intended pre-combined packet. The compressed CDC scheme exploits both the compression opportunities within individual functions, and the multicasting opportunities across different functions, and achieves a smaller communication load of $\frac{3}{2}$.

III. Problem Formulation and Main Results

We consider a computation job of processing N input files, for some $N \in \mathbb{N}$, to compute Q output functions, for some $Q \in \mathbb{N}$. We denote the input files as $w_1, \ldots, w_N \in \mathbb{F}_{2^F}$, for some $F \in \mathbb{N}$, and the output functions as $\phi_1, \ldots, \phi_Q : (\mathbb{F}_{2^F})^N \to \mathbb{F}_{2^T}$, for some $T \in \mathbb{N}$. We focus on the computation jobs with *linear* aggregation for which the computation of each output function can be decomposed as the sum of N intermediate values computed from the input files, i.e., for $q = 1, \ldots, Q$,

$$\phi_q(w_1, \ldots, w_N) = v_{q,1} + v_{q,2} \cdots + v_{q,N}, \qquad (1)$$

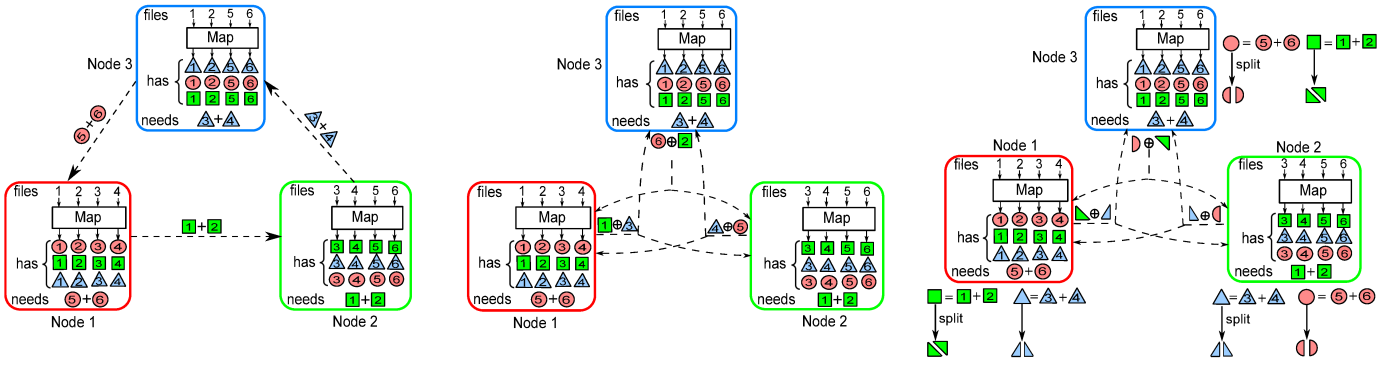

Fig. 2: Coded computing schemes for a MapReduce job with linear Reduce functions, which processes 6 files to compute 3 functions, over 3 distributed computing nodes each with a storage size of 4 files.

where $v_{q,n} = g_q(w_n)$ is the intermediate value of ϕ_q computed from some intermediate function $g_q : \mathbb{F}_{2^F} \to \mathbb{F}_{2^T}$. So far, we have introduced one computation job that involves computing Q functions. Here, we consider the scenario where J such computation jobs are executed in parallel, for some $J \in \mathbb{N}$. We denote the N input files of job j as $w_{1^{(j)}}, \ldots, w_{N^{(j)}}$, and the Q output functions job j wants to compute as $\phi_{1^{(j)}}, \ldots, \phi_{Q^{(j)}}$.[1]

A. Network model

The above described J computation jobs are executed distributedly on a computer cluster that consists of K distributed computing nodes, for some $K \in \mathbb{N}$. These computing nodes are denoted as Node $1, \ldots,$ Node K. Here we assume $K \leq N$, and focus on a symmetric setting for the sake of load balancing, in which $K|Q$, and each node is responsible for computing $\frac{Q}{K}$ output functions for each job. The K nodes are connected through an error-free broadcast network. Each node has a local memory that can store up to μJN input files, i.e., μ fraction of the entire dataset that contains all input files from all jobs, for some μ satisfying $\frac{1}{K} \leq \mu < 1$.

Before the computation starts, each node selects and stores μJN input files from the dataset. For each node k, we denote the set of indices of the files stored locally as $\mathcal{M}_k$. A valid file placement has to satisfy 1) $|\mathcal{M}_k| \leq \mu JN$, for all $k = 1, 2, \ldots, K$ (local storage constraint), and 2) $\cup_{k=1,\ldots,K} \mathcal{M}_k = \cup_{j=1,\ldots,J} \{n^{(j)} : n = 1, 2, \ldots, N\}$ (the entire dataset needs to be collectively stored across the cluster).

B. Distributed computing model

The computation follows a MapReduce model that consists of three phases: Map phase, Shuffle phase, and Reduce phase.

Map phase. For each file $w_{n^{(j)}}$ of job j, $n^{(j)} \in \mathcal{M}_k$, Node k maps it into Q intermediate values $v_{1^{(j)},n^{(j)}}, v_{2^{(j)},n^{(j)}}, \ldots, v_{Q^{(j)},n^{(j)}}$, one for each of the Q functions computed in job j. We assume that all the intermediate values across the J jobs have the same size of T bits, which is the case when for example, we are training J image classifiers in parallel using the same deep neural network.

Shuffle phase. For each computation job j, we assign the tasks of reducing the output functions symmetrically across the nodes, such that each node computes a disjoint subset of $\frac{Q}{K}$ functions. We denote the set of the indices of the output functions assigned to Node k for job j as $\mathcal{S}_k^{(j)}$, $j = 1, 2, \ldots, J$.

In the Shuffle phase, each node k produces a message, denoted by $X_k \in \mathbb{F}_{2^{\ell_k}}$, as a function of the locally computed Map results, where $\ell_k \in \mathbb{N}$ denotes the length of the message in bits, and broadcasts X_k to all other nodes.

Definition 1 (Communication Load)**.** We define the *communication load*, denoted by L, as the total number of bits contained in all broadcast messages, normalized by JQT, i.e.,

$$L \triangleq \frac{\ell_1 + \cdots + \ell_K}{JQT}. \tag{2}$$

Reduce phase. For each job j and each $q^{(j)} \in \mathcal{S}_k^{(j)}$, $j = 1, 2, \ldots, J$, Node k computes the output function $\phi_{q^{(j)}}$ as in (1), using the locally computed Map results and the received broadcast messages in the Shuffle phase.

C. Main Results

For the above formulated distributed computing problem, we first study the effects of applying the compression scheme and the CDC scheme individually on reducing the communication load. Then, we present our main result, which is a communication load achieved by the proposed computing scheme that jointly utilizes compression and CDC.

Exploiting the compression technique, each sender node pre-combines all the intermediate values needed at the receiver node for a particular function, and then sends the pre-combined value. We demonstrate, in the appendix of the full version of this paper [23], that the following communication load can be achieved by solely applying compression.

$$L_{\text{compression}} = \lceil \tfrac{1}{\mu} \rceil - 1, \quad \tfrac{1}{K} \leq \mu < 1. \tag{3}$$

When only applying the CDC scheme without compression, as shown in [6], we can achieve the communication load

$$L_{\text{CDC}} = \frac{(1-\mu)N}{\mu K}. \tag{4}$$

The CDC scheme creates coded multicast packets that are simultaneously useful for μK nodes. Hence, for fixed storage size μ, the achieved communication load L_{CDC} decreases inversely proportionally with the network size (K). On the other hand, since the CDC scheme was designed to handle general Reduce functions that require each of the N intermediate values separately as the inputs, L_{CDC} also scales linearly with the number of input files (N).

[1] For example, we can consider executing J machine learning tasks (e.g., image classification) in parallel, each of which has its own dataset, and aims to obtain its own set of model parameters.

We propose the compressed coded distributed computing (compressed CDC) scheme, which jointly utilizes the combining and the coded multicasting techniques. We present the performance of compressed CDC in the following theorem, and give its general description in the next section

Theorem 1. *To execute J computation jobs with linear aggregation of intermediate results, each of which processes N input files to compute Q output functions, distributedly over K computing nodes each with a local storage of size μ, the proposed compressed CDC scheme achieves the following communication load*

$$L_{\text{compressed CDC}} = \frac{(1-\mu)(\mu K+1)}{\mu K}, \tag{5}$$

for $\mu K \in \{1,\ldots,K-1\}$, and $J = \gamma\binom{K}{\mu K+1}$, for some $\gamma \in \mathbb{N}$.

Remark 1. Compared with the compression scheme whose communication load is in (3), for large K, the proposed compressed CDC scheme reduces the communication load by a factor of μ when $\frac{1}{K} \leq \mu < \frac{1}{2}$, and by a factor of $1-\mu$ when $\frac{1}{2} \leq \mu < 1$. In the scenarios where the cluster consists of many low-end computing nodes with small storage size (e.g., $\mu = \frac{1}{K}$), this reduction factor can scale with the network size. Also, in contrast to the compression scheme, the load $L_{\text{compressed CDC}}$ keeps decreasing as storage size μ increases. □

Remark 2. Unlike the communication load in (4) achieved by the CDC scheme, the load achieved by the compressed CDC scheme does not grow with the number of input files. This is accomplished by pre-combining multiple intermediate values of the same Reduce function. □

Remark 3. The file placement of the compressed CDC scheme is performed such that all N input files of each particular computation job are placed exclusively on a unique subset of $\mu K + 1$ nodes, following a repetitive pattern specified by the CDC scheme. As a result, the compressed CDC scheme executes a batch of $\binom{K}{\mu K+1}$ jobs in parallel. In the Shuffle phase of compressed CDC, each computing node first pre-combines several intermediate values of a single function reduced at another node, and then applies bit-wise XOR operations on multiple such pre-combined packets to generate a coded multicast packet that is simultaneously useful for computing μK functions. We note that these μK functions can be different functions in the same job, as well as different functions in different jobs. □

IV. Description of the compressed CDC scheme

We present the general compressed CDC scheme. In [23], we also provide a running example to illustrate the scheme.

We consider the storage size μ such that $\mu K \in \{1,2,\ldots,K-1\}$, and take sufficiently many computation jobs to process in parallel, where the number of jobs $J = \gamma\binom{K}{\mu K+1}$, for some $\gamma \in \mathbb{N}$. The proposed compressed CDC scheme operates on a batch of $\binom{K}{\mu K+1}$ jobs at a time, and repeats the same operations γ times to process all jobs. Therefore, it is sufficient to describe the scheme for the case of $\gamma = 1$.

A. File placement

For each job j, $j = 1,2,\ldots,\binom{K}{\mu K+1}$, all of its input files $w_{1^{(j)}}, w_{2^{(j)}}, \ldots, w_{N^{(j)}}$ are stored exclusively on a unique subset of $\mu K + 1$ nodes, and we denote the set of indices of these nodes as $\mathcal{K}_j$. Within $\mathcal{K}_j$, each file $w_{n^{(j)}}$ of job j is repeatedly stored on μK nodes. In particular, we first evenly partition the files $w_{1^{(j)}}, w_{2^{(j)}}, \ldots, w_{N^{(j)}}$ into $\mu K+1$ batches, and label each batch by a unique size-μK subset of $\mathcal{K}_j$, denoted by $\mathcal{P}_j$. Then, we store all the files in a batch on each of the μK nodes whose index is in the corresponding subset $\mathcal{P}_j$. We denote the set of indices of the files from job j in a batch labelled by a subset $\mathcal{P}_j$ as $\mathcal{B}_{\mathcal{P}_j}$. The file placement is performed such that for each $\mathcal{P}_j \subset \mathcal{K}_j$ with $|\mathcal{P}_j| = \mu K$, and each $n^{(j)} \in \mathcal{B}_{\mathcal{P}_j}$, we have $n^{(j)} \in \mathcal{M}_k$, for all $k \in \mathcal{P}_j$, where $\mathcal{M}_k$ is the set of indices of all files stored at Node k.

Applying the above file placement, each node in $\mathcal{K}_j$ stores $\mu K \times \frac{N}{\mu K+1}$ files. Since each node is in $\binom{K-1}{\mu K}$ subsets of $\{1,2,\ldots,K\}$ of size $\mu K + 1$, it stores overall $\frac{\mu K N}{\mu K+1} \times \binom{K-1}{\mu K} = \mu J N$ files, satisfying its local storage constraint.

B. Coded computing

The compressed CDC scheme performs computation and data shuffling in subsets of $\mu K+1$ nodes. Within each subset $\mathcal{K}_j$, $j = 1,2,\ldots,\binom{K}{\mu K+1}$, that contains the indices of $|\mathcal{K}_j| = \mu K+1$ nodes, the scheme proceeds in two stages. In the first stage, the nodes in $\mathcal{K}_j$ process the files they have exclusively stored, i.e., the files of job j. In the second stage, they handle the files from other jobs.

1) Stage 1 (coding for a single job): All computing nodes in $\mathcal{K}_j$ only process input files and compute output functions for job j. For ease of exposition, we drop all the job indices in the rest of the description of stage 1.

In the Map phase, each node $k \in \mathcal{K}$ maps all the files of job j it has stored locally, for all output functions of job j. After the Map phase, for each subset $\mathcal{P}$ of size μK, and $k' \in \mathcal{K}\backslash\mathcal{P}$, each node in $\mathcal{P}$ has computed $\frac{Q}{K}$ intermediate values, one for each of the functions assigned to Node k', from each of the files in the batch $\mathcal{B}_{\mathcal{P}}$. More precisely, these intermediate values are $\{v_{q,n} : q \in \mathcal{S}_{k'}, n \in \mathcal{B}_{\mathcal{P}}\}$.

In the Shuffle phase, for each subset $\mathcal{P} \subset \mathcal{K}$ of size μK, the nodes in $\mathcal{P}$ sum up the above intermediate values, for each $q \in \mathcal{S}_{k'}$, obtaining a pre-combined value $\bar{v}_{q,\mathcal{P}} = \sum_{n\in\mathcal{B}_{\mathcal{P}}} v_{q,n}$.

Having computed $\frac{Q}{K}$ such pre-combined values $\{\bar{v}_{q,\mathcal{P}} : q \in \mathcal{S}_{k'}\}$, the nodes in $\mathcal{P}$ concatenate them to generate a packet $V_{\mathcal{P}}$, and evenly and arbitrarily split it into μK segments. We label the segments by the elements in $\mathcal{P}$. That is, for $\mathcal{P} = \{i_1, i_2, \ldots, i_{\mu K}\}$, we have $V_{\mathcal{P}} = (V_{\mathcal{P},i_1}, V_{\mathcal{P},i_2}, \ldots, V_{\mathcal{P},i_{\mu K}})$.

Finally, each node k in $\mathcal{K}$ generates a coded packet $X_k^{\text{stage 1}}$ by computing bit-wise XOR (denoted by $\oplus$) of the data segments labelled by k, i.e.,

$$X_k^{\text{stage 1}} = \underset{\mathcal{P}\subset\mathcal{K}:|\mathcal{P}|=\mu K, k\in\mathcal{P}}{\oplus} V_{\mathcal{P},k}, \tag{6}$$

and multicasts X_k to all other nodes in $\mathcal{K}$.

After Node k receives a coded packet $X_{k'}^{\text{stage 1}}$ from Node k', it cancels all the segments $V_{\mathcal{P},k'}$s with $k \in \mathcal{P}$, and recovers the intended segment $V_{\mathcal{K}\backslash\{k\},k'}$. Repeating this decoding process for all received coded packets, Node k recovers $V_{\mathcal{K}\backslash\{k\}}$, and hence $\bar{v}_{q,\mathcal{K}\backslash\{k\}}$, for all $q \in \mathcal{S}_k$. Using these values, together with the local Map results, Node k computes the output ϕ_q

for all $q \in \mathcal{S}_k$. After the first stage of computation, each node in $\mathcal{K}_j$ completes its computation tasks for job j.

Since each of the coded packets in (6) contains $\frac{Q}{K} \times \frac{T}{\mu K}$ bits, the communication load exerted in the Shuffle phase of the first stage is $L_{\text{stage 1}} = \frac{\frac{Q}{K} \times \frac{(\mu K+1)T}{\mu K}}{JQT} = \frac{\frac{\mu K+1}{\mu K}}{JK}$.

2) Stage 2 (coding across jobs): For a node i outside $\mathcal{K}_j$, and each $k \in \mathcal{K}_j$, we label the job whose input files are exclusively stored on the nodes in $\{i\} \cup \mathcal{K}_j \backslash \{k\}$ as j_k. The nodes in $\mathcal{P}_{j_k} = \mathcal{K}_j \backslash \{k\}$ share a batch of $\frac{N}{\mu K+1}$ files in $\mathcal{B}_{\mathcal{P}_{j_k}}$ for job j_k. In the Map phase, for each $k' \in \mathcal{P}_{j_k}$, Node k' computes $\frac{Q}{K}$ intermediate values, one for each function of job j_k assigned to Node k in $\mathcal{S}_k^{(j_k)}$, from each of the files in the batch $\mathcal{B}_{\mathcal{P}_{j_k}}$. More precisely, these intermediate values are $\{v_{q^{(j_k)},n^{(j_k)}} : q^{(j_k)} \in \mathcal{S}_k^{(j_k)}, n^{(j_k)} \in \mathcal{B}_{\mathcal{P}_{j_k}}\}$.

In the Shuffle phase, for each $k \in \mathcal{K}_j$, the nodes in $\mathcal{P}_{j_k}$ sum up the above Map results, for each $q^{(j_k)} \in \mathcal{S}_k^{(j_k)}$, obtaining $\bar{v}_{q^{(j_k)},\mathcal{P}_{j_k}} = \sum_{n^{(j_k)} \in \mathcal{B}_{\mathcal{P}_{j_k}}} v_{q^{(j_k)},n^{(j_k)}}$.

Next, as similarly done in the first stage, the nodes in $\mathcal{P}_{j_k}$ first concatenate the above $\frac{Q}{K}$ pre-combined values $\{\bar{v}_{q^{(j_k)},\mathcal{P}_{j_k}} : q^{(j_k)} \in \mathcal{S}_k^{(j_k)}\}$ to generate a packet $V_{\mathcal{P}_{j_k}}$, and then split it into μK segments. We label these segments by the elements in $\mathcal{P}_{j_k}$, i.e., for $\mathcal{P}_{j_k} = \{i_1, i_2, \ldots, i_{\mu K}\}$, we have $V_{\mathcal{P}_{j_k}} = (V_{\mathcal{P}_{j_k},i_1}, V_{\mathcal{P}_{j_k},i_2}, \ldots, V_{\mathcal{P}_{j_k},i_{\mu K}})$.

Finally, each node k' in $\mathcal{K}_j$ generates a coded packet $X_{k'}^{\text{stage 2}}$ by bit-wise XORing the data segments labelled by k', i.e.,

$$X_{k'}^{\text{stage 2}} = \underset{t \in \mathcal{K}_j \backslash \{k'\}}{\oplus} V_{\mathcal{P}_{j_t},k'}, \tag{7}$$

and multicasts $X_{k'}^{\text{stage 2}}$ to all other nodes in $\mathcal{K}_j$.

We note that since the job index j_t (whose input files are exclusively stored on nodes in $\{i\} \cup \mathcal{K}_j \backslash \{t\}$) is different for different t, the above coded packet is generated using intermediate values from different jobs.

Having received a coded packet $X_{k'}^{\text{stage 2}}$ from Node k', Node k cancels all the segments $V_{\mathcal{P}_{j_t},k'}$s with $k \in \mathcal{P}_{j_t}$, and recovers the intended segment $V_{\mathcal{P}_{j_k},k'}$. Repeating this decoding process for all received coded packets, Node k recovers $V_{\mathcal{P}_{j_k}}$, and hence $\bar{v}_{q^{(j_k)},\mathcal{P}_{j_k}}$, for all $q^{(j_k)} \in \mathcal{S}_k^{(j_k)}$.

We repeat the above operations for all $i \in \{1, 2, \ldots, K\} \backslash \mathcal{K}_j$. By the end of the second stage, each node in $\mathcal{K}_j$ recovers partial sums to compute functions from $K - \mu K - 1$ jobs.

The communication load incurred in the Shuffle phase, for a particular i, is $\frac{\frac{Q}{K} \times \frac{\mu K+1}{\mu K}}{JQ}$, and the total communication load of the second stage is $L_{\text{stage 2}} = \frac{(K-\mu K-1)\frac{\mu K+1}{\mu K}}{JK}$.

Having performed this two-stage operation on all subsets $\mathcal{K}_j$ of $\mu K+1$ nodes, $j = 1, 2, \ldots, \binom{K}{\mu K+1}$, each node k finishes computing its assigned functions from $\binom{K-1}{\mu K}$ jobs. For each of the remaining $\binom{K}{\mu K+1} - \binom{K-1}{\mu K}$ jobs, say job j', and each $k' \in \mathcal{K}_{j'}$, Node k receives a partial sum of $\frac{N}{\mu K+1}$ intermediate values for each of the functions in $\mathcal{S}_k^{(j')}$, in the subset $\{k\} \cup \mathcal{K}_{j'} \backslash \{k'\}$. Summing up these $\mu K + 1$ partial sums, Node k finishes computing each of its assigned functions from job j'.

The overall communication load of compressed CDC is $L_{\text{compressed CDC}} = \binom{K}{\mu K+1} \times (L_{\text{stage 1}} + L_{\text{stage 2}}) = \frac{(1-\mu)(\mu K+1)}{\mu K}$.

V. Acknowledgement

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This work is also in part supported by ONR award N000141612189 and NSF Grants CCF-1703575 and NeTS-1419632.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Sixth USENIX OSDI*, Dec. 2004.

[2] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," *NIPS*, pp. 693–701, 2011.

[3] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, Aug. 2011.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *IEEE CVPR*, pp. 770–778, 2016.

[5] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," *53rd Allerton Conference*, Sept. 2015.

[6] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *IEEE Trans. Inf. Theory*, vol. 64, no. 1, Jan. 2018.

[7] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded terasort," *IPDPS ParLearning Workshop*, May 2017.

[8] S. Li, Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "A scalable framework for wireless distributed computing," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 2643–2654, Oct. 2017.

[9] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded distributed computing: Straggling servers and multistage dataflows," *54th Allerton Conference*, Sept. 2016.

[10] Y. H. Ezzeldin, M. Karmoose, and C. Fragouli, "Communication vs distributed computation: an alternative trade-off curve," *e-print arXiv:1705.08966*, 2017.

[11] M. Kiamari, C. Wang, and A. S. Avestimehr, "On heterogeneous coded distributed computing," *IEEE GLOBECOM*, Dec. 2017.

[12] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," *Interspeech*, 2014.

[13] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *NIPS*, pp. 1707–1718, 2017.

[14] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.

[15] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," *NIPS*, pp. 2100–2108, 2016.

[16] R. Tandon, Q. Lei, A. Dimakis, and N. Karampatziakis, "Gradient coding," *NIPS Machine Learning Systems Workshop*, 2016.

[17] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," *IEEE NetCod*, Dec. 2016.

[18] ——, "Coding for distributed fog computing," *IEEE Commun. Mag.*, vol. 55, no. 4, Apr. 2017.

[19] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," *NIPS*, pp. 4406–4416, 2017.

[20] ——, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," *e-print arXiv:1801.07487*, 2018.

[21] L. Song, C. Fragouli, and T. Zhao, "A pliable index coding approach to data shuffling," *IEEE ISIT*, pp. 2558–2562, 2017.

[22] M. A. Attia and R. Tandon, "Information theoretic limits of data shuffling for distributed learning," *IEEE GLOBECOM*, Dec. 2016.

[23] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Compressed coded distributed computing," *e-print arXiv:1805.01993*, 2018.