Speeding Up Distributed Gradient Descent by Utilizing Non-persistent Stragglers

Emre Ozfatura[†], Deniz Gündüz[†] and Sennur Ulukus[‡]
[†]Information Processing and Communications Lab, Dept. of Electrical and Electronic Engineering, Imperial College London
[‡]Department of Electrical and Computer Engineering, Institute for Systems Research, University of Maryland, College Park
{m.ozfatura,d.gunduz}@imperial.ac.uk, ulukus@umd.edu

***Abstract*—When gradient descent (GD) is scaled to many parallel computing servers (workers) for large scale machine learning problems, its per-iteration computation time is limited by the *straggling* workers. Coded distributed GD (DGD) can tolerate straggling workers by assigning redundant computations to the workers, but in most existing schemes, each non-straggling worker transmits one message per iteration to the parameter server (master) after completing all its computations. We allow multiple computations to be conveyed from each worker per iteration in order to exploit computations executed also by the straggling worker. We show that the average completion time per iteration can be reduced significantly at a reasonable increase in the communication load. We also propose a general coded DGD technique which can trade-off the average computation time with the communication load.**

I. Introduction

For given N training data points $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N]^T$, $\mathbf{x}_i \in \mathbb{R}^d$, and the corresponding labels $\mathbf{y} = [y_1, \dots, y_N]^T$, $y_i \in \mathbb{R}$, $i \in [N] \triangleq \{1, 2, \dots, N\}$, the objective of many machine learning problems is to minimize the *parameterized empirical loss function*

$$L(\boldsymbol{\theta}) \triangleq \sum_{i=1}^{N} l\left((\mathbf{x}_i, y_i), \boldsymbol{\theta}\right) + \lambda R(\boldsymbol{\theta}), \tag{1}$$

where $\boldsymbol{\theta} \in \mathbb{R}^d$ is the parameter vector, l is an application specific loss function, and $R(\boldsymbol{\theta})$ is the regularization component. This optimization problem is commonly solved by gradient descent (GD), where at each iteration, the parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$ is updated opposite of the GD direction:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}), \tag{2}$$

where η_t is the learning rate at iteration t, and the gradient at the current parameter vector is given by $\nabla_{\boldsymbol{\theta}} = \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} l\left((y_i, x_i), \boldsymbol{\theta}\right)$.

When $\mathbf{X}$ is a large dataset, as in many problems, GD approach may require large computation time. To this end, a parallel computation framework can be utilized to reduce the convergence time [1]. Hence, the gradient computation task is divided into smaller sub-tasks of computing a partial gradient over a subset of the dataset (often referred as mini-batch), which are distributed across multiple workers to be executed in parallel. Each worker sends its local gradient estimate back to the parameter server, which will be referred as the *master*, as soon as it is computed. When all the local gradient estimates are aggregated by the master, their average is used to update $\boldsymbol{\theta}_{t+1}$, as in (2), which is then transmitted to all the workers for the next iteration. While distributed computation is essential to handle large data sets, the completion time of each iteration is constrained by the slowest worker(s), called the *straggling worker(s)*, which can be detrimental for the convergence of the algorithm. The randomness of the straggling workers can be considered to model a packet erasure communication channel, in which the transmitted data packets are randomly erased [2]. Motivated by this analogy, several papers have recently introduced coding theoretic ideas in order to mitigate the effect of straggling workers in DGD [2]–[10].

The key limitation of the aforementioned works is that straggling behaviour of the workers is treated as all or nothing (straggler/non-straggler), and the computations of the straggler workers are discarded as long as they cannot complete all the assigned computations. This leads to the underutilization of the computational resources. Moreover, in practice, *non-persistent* stragglers may complete a significant portion of their assigned tasks.

Therefore, our main objective in this paper is to redesign the straggling avoidance techniques in a way that computational capacity of the non-persistent stragglers can also be utilized. This will be achieved by allowing each worker to send multiple messages to master at each iteration, which we refer to as *multi-message communication (MMC)*. We define the average number of computations conveyed to the master from the workers as the *communication load*, and show that there is a trade-off between the communication load and the computation time. In this paper, we aim to answer three main questions regarding the straggler avoidance: How con we redesign coded computation strategy [4], [7] to utilize non-persistent stragglers? How can we balance the increase in the communication load with the computation time? When is it better to use coded/uncoded computation?

Various other distributed computation techniques with MMC have recently been considered in [11] and [12]; however, their analyses are limited to specific problem setups where the main computation problem boils down to matrix-vector multiplication. Moreover, they focus exclusively on the computation time,

This work was supported by EC H2020-MSCA-ITN-2015 project SCAVENGE under grant number 675891, and by the European Research Council project BEACON under grant number 677854.

and the communication-computation trade-off is not explicitly illustrated.

II. Coded Computation

Recently particular attention has been paid to the least squares linear regression problem, where the gradient computation reduces to a linear matrix operation, i.e., $\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}_t - \mathbf{X}^T\mathbf{y}$. For this particular problem each iteration of DGD can be treated as a distributed matrix-matrix or matrix-vector multiplication[1] and dataset is encoded before being distributed to workers to avoid stragglers.

Another straggler avoidance strategy, which is applicable to more general DGD problems, is *gradient coding* [8]–[10] where each worker computes multiple partial gradients over original dataset but sends the linear combination of those partial gradients to the master.

For the coded computation framework, consider a distributed architecture with M workers, $\mathrm{W}_1, \ldots, \mathrm{W}_M$. Dataset $\mathbf{X}$ is divided into M submatrices each of size $N/M \times d$. Then, these submatrices are encoded and distributed across the workers such that coded submatrices $\tilde{\mathbf{x}}_i^{(1)}, \ldots, \tilde{\mathbf{x}}_i^{(r)}$, each size of $N/M \times d$, are assigned to W_i to compute $\tilde{\mathbf{x}}_i^{(1)}(\tilde{\mathbf{x}}_i^{(1)})^T\boldsymbol{\theta}_t, \ldots, \tilde{\mathbf{x}}_i^{(r)}(\tilde{\mathbf{x}}_i^{(r)})^T\boldsymbol{\theta}_t$. We refer to the number of computations assigned to each worker, r, as the computation load.

Once all these computations are executed, W_i returns their sum to the master. The results obtained from a sufficient number of workers are used at the master to evaluate $\boldsymbol{\theta}_{t+1}$. Now we will briefly summarize the Lagrange coded computation method introduced in [4], [7], which utilizes polynomial interpolation for the code design[2].

A. Lagrange Polynomial

Consider the following vector-valued polynomial

$$f(z) \triangleq \sum_{i\in[N]} \mathbf{a}_i \prod_{j\in[N]\setminus\{i\}} \frac{z-\alpha_j}{\alpha_i-\alpha_j}, \tag{3}$$

where $\alpha_1, \ldots, \alpha_N$ are N distinct real numbers, and $\mathbf{a}_1, \ldots, \mathbf{a}_N$ are vectors of size $1 \times k$. We have $f(\alpha_i) = \mathbf{a}_i$, $\forall i \in [N]$. Let us consider another polynomial $h(z) = f(z)f(z)^T\boldsymbol{\theta}_t$, such that $h(\alpha_i) = \mathbf{a}_i\mathbf{a}_i^T\boldsymbol{\theta}_t$. Hence, if the coefficients of polynomial $h(z)$ are known, then the term $\sum_{i=1}^N \mathbf{a}_i\mathbf{a}_i^T\boldsymbol{\theta}_t$ can be obtained easily from $\mathbf{a}_1, \ldots, \mathbf{a}_N$. We remark that the degree of the polynomials $f(z)$ and $h(z)$ are $N-1$ and $2N-2$, respectively. Accordingly, if the value of $h(z)$ at $2N-1$ distinct points are known at the master, then $h(z)$ can be recovered via polynomial interpolation. This is the key notion behind Lagrange coded computation [4], which is explained in the next subsection.

B. Lagrange Coded Computation (LCC)

Let us first assume that N is a multiple of r. For given r and N, the rows of $\mathbf{X}$, $\mathbf{x}_1, \ldots, \mathbf{x}_N$, are divided into r disjoint groups, each of size N/r, and the rows within each group are ordered according to their indices. Let $\mathbf{x}_{k,j}$ denote the jth row in the kth group, and $\mathbf{X}_k$ denote all the rows in the kth group; that is, $\mathbf{X}_k$ is the $N/r \times d$ submatrix of $\mathbf{X}$. Then, for distinct real numbers $\alpha_1, \ldots, \alpha_{N/r}$, we form the following r structurally identical polynomials of degree $N/r-1$, taking the rows of $\mathbf{X}_k$ as their coefficients:

$$f_k(z) = \sum_{i=1}^{N/r} \mathbf{x}_{k,i} \prod_{j=1, j\neq i}^{N/r} \frac{z-\alpha_j}{\alpha_i-\alpha_j}, \ k \in [r]. \tag{4}$$

Then, we define

$$H(z) \triangleq \sum_{k=1}^{r} f_k(z)f_k(z)^T\boldsymbol{\theta}_t. \tag{5}$$

Coded vectors $\tilde{\mathbf{x}}_i^{(k)}$, $k \in [r]$, for W_i, $i \in [N]$ are obtained by evaluating $f_k(z)$ polynomials at distinct values, $\beta_i \in \mathbb{R}$, i.e., $\tilde{\mathbf{x}}_i^{(k)} = f(\beta_i)$. At each iteration of the DGD algorithm W_i returns the value of

$$H(\beta_i) = \sum_{k=1}^{r} \tilde{\mathbf{x}}_i^{(k)}(\tilde{\mathbf{x}}_i^{(k)})^T\boldsymbol{\theta}_t. \tag{6}$$

The degree of polynomial $H(z)$ is $2N/r-2$; and thus, the non-straggling threshold for LCC is given by $K_{LCC}(r) = 2N/r-1$; that is, having received the value of $H(z)$ at $K_{LCC}(r)$ distinct points, the master can extrapolate $H(z)$ and compute

$$\sum_{j=1}^{N/r} H(\alpha_j) = \mathbf{X}^T\mathbf{X}\boldsymbol{\theta}_t, \tag{7}$$

When N is not divisible by r, zero-valued data points can be added to $\mathbf{X}$ to make it divisible by r. Hence, in general the non-straggling threshold is given by $K_{LCC}(r) = 2\lceil N/r\rceil - 1$.

C. LCC with MMC

Here, we introduce LCC with MMC by using a single polynomial $f(z)$ of degree $N-1$, instead of using r different polynomials each of degree $N/r-1$. We define

$$f(z) \triangleq \sum_{i=1}^{N} \mathbf{x}_i \prod_{j=1, j\neq i}^{N} \frac{z-\alpha_j}{\alpha_i-\alpha_j}, \tag{8}$$

where $\alpha_1, \ldots, \alpha_N$ are N distinct real numbers, and we construct

$$h(z) \triangleq f(z)f(z)^T\boldsymbol{\theta}_t, \tag{9}$$

such that $h(\alpha_i) = \mathbf{x}_i\mathbf{x}_i^T\boldsymbol{\theta}_t$. Consequently, if the polynomial $h(z)$ is known at the master, then the full gradient $\sum_{i=1}^N h(\alpha_i) = \sum_{i=1}^N \mathbf{x}_i\mathbf{x}_i^T\boldsymbol{\theta}_t$ can be obtained. To this end, r coded vectors $\tilde{\mathbf{x}}_i^{(1)}, \ldots, \tilde{\mathbf{x}}_i^{(r)}$, which are assigned to W_i, $i \in [N]$ are constructed by evaluating $f(z)$ at r different points, $\beta_i^{(1)}, \ldots, \beta_i^{(r)}$, i.e.,

$$\tilde{\mathbf{x}}_i^{(j)} = f(\beta_i^{(j)}), \ i \in [N], j \in [r]. \tag{10}$$

W_i computes $\tilde{\mathbf{x}}_i^{(1)}(\tilde{\mathbf{x}}_i^{(1)})^T\boldsymbol{\theta}_t, \ldots, \tilde{\mathbf{x}}_i^{(r)}(\tilde{\mathbf{x}}_i^{(r)})^T\boldsymbol{\theta}_t$, and transmits the resultant vector to the master after each computation. Coded computation corresponding to coded data point $\tilde{\mathbf{x}}_i^{(j)}$ at W_i provides the value of polynomial $h(z)$ at point $\beta_i^{(j)}$. The degree of the polynomials $f(z)$ and $h(z)$ are $N-1$ and $2(N-1)$,

[1] When $\mathbf{Q} = \mathbf{X}^T\mathbf{X}$ is known in advance, the computation task reduces to evaluating $\mathbf{Q}\boldsymbol{\theta}_t$.

[2] Throughout the paper we will assume $M = N$ for simplicity, although all these methods can be generalized to any M, N pair.

respectively, which implies that $h(z)$ can be interpolated from its values at any $2N-1$ distinct points. Hence, any $2N-1$ computations received from any subset of the workers are sufficient to obtain the full gradient.

We note that, in the original LCC scheme coded data points are constructed evaluating r different polynomials at the same data point, whereas in the multi-message LCC scheme, coded data points are constructed evaluating a single polynomial at r distinct points. Per iteration completion time can be reduced with MMC since the partial computations of the non-persistent stragglers are also utilized; however, at the expense of an increase in the communication load. Nevertheless, it is possible to set the number of polynomials to a different value to seek a balance between the communication load and the per iteration completion time. This will be illustrated in Section V.

III. Uncoded Computation and Communication (UCUC)

In UCUC, the data points are divided into N groups, where N is the number of workers, and each group is assigned to a different worker. While the per iteration completion time is determined by the slowest worker in this case, it can be reduced by assigning multiple data points to each worker, and allowing it to communicate the result of its computation for each data point right after its execution. We also remark here that, with UCUC the master can apply SGD, and evaluate the next iteration of the parameter vector without waiting for all the computations, which is not possible with coded computation. While we will mainly consider GD with a full gradient computation in our analysis for a fair comparison with the presented coded DGD approaches, we will show in Section V that significant gains can be obtained in both computation time and communication load by ignoring only 5% of the computations.

Let $\mathbf{A}$ be the assignment matrix for the data points to workers, where $\mathbf{A}(j,k)=i$ means that the ith data point is computed by the kth worker in the jth order. An efficient way of constructing $\mathbf{A}$ is to use a circular shift matrix, where

$$\mathbf{A}(j,:) = \text{circshift}\ ([1:N], -(j-1)). \tag{11}$$

We also note that non-persistent stragglers can be also utilized via local parameter updating and weighted averaging over workers without MMC [13].

IV. Per Iteration Completion Time Statistics

In this section, we analyze the statistics of per iteration completion time T for the DGD schemes introduced above. For the analysis we consider a setup with N workers and assume that the dataset is divided into N data points (these could also be mini-batches). For the straggling behavior, we adopt the model in [2] and [11], and assume that the probability of completing s computations at any worker by time t is given by

$$F_s(t) \triangleq \begin{cases} 1-e^{-\mu(\frac{t}{s}-\alpha)}, & \text{if } t \geq s\alpha, \\ 0, & else. \end{cases} \tag{12}$$

The statistical model considered above is a shifted exponential distribution, such that the duration of a computation cannot be less than α. Under this model, although the overall computation time at a particular worker has an exponential distribution, the duration of each computation is assumed to be identical. Further, let $P_s(t)$ denote the probability of completing exactly s computations by time t. We have $F_s(t)=\sum_{s'=s}^{r} P_{s'}(t)$, where $P_r(t)=F_r(t)$, since there are a total of r computations assigned to each user. One can observe that $P_s(t)=F_s(t)-F_{s+1}(t)$; and, hence $P_s(t)$ can be written as follows:

$$P_s(t) = \begin{cases} 0, & \text{if } t < s\alpha, \\ 1-e^{-\mu(\frac{t}{s}-\alpha)}, & s\alpha \leq t < (s+1)\alpha. \\ e^{-\mu(\frac{t}{s+1}-\alpha)} - e^{-\mu(\frac{t}{s}-\alpha)}, & (s+1)\alpha < t, \end{cases} \tag{13}$$

We divide the workers into $r+1$ groups according to the number of computations completed by time t. Let $N_s(t)$ be the number of workers that have completed exactly s computations by time t, $s=0,\dots,r$, and define $\mathbf{N}(t) \triangleq (N_0(t),\dots,N_r(t))$, where $\sum_{s=0}^{r} N_s(t)=N$. The probability of a particular realization is given by

$$\Pr(\mathbf{N}(t)) = \prod_{s=0}^{r} P_s(t)^{N_s} \binom{N-\sum_{j<s} N_j}{N_s}. \tag{14}$$

At this point, we introduce $M(t)$, which denotes the total number of computations completed by all the workers by time t, i.e., $M(t) \triangleq \sum_{s=1}^{r} s \times N_s(t)$, and let M_{th} denote the threshold for obtaining the full gradient. Hence, the probability of recovering the full gradient at the master by time t, $\Pr(T \leq t)$, is given by $\Pr(M(t) \geq M_{th})$. Consequently, we have

$$\Pr(T \leq t) = \sum_{\mathbf{N}(t):M(t)\geq M_{th}} \Pr(\mathbf{N}(t)), \tag{15}$$

and

$$E[T] = \int_0^\infty \Pr(T>t)dt \tag{16}$$

$$= \int_0^\infty \left[1 - \sum_{\mathbf{N}(t):M(t)\geq M_{th}} \Pr(\mathbf{N}(t))\right] dt. \tag{17}$$

Per iteration completion time statistics of non-straggler threshold based schemes can be derived similarly. For a given non-straggler threshold K_{th}, and per worker computation load r, we have

$$\Pr(T \leq t) = \sum_{k=K_{th}}^{N} \binom{N}{k} (1-e^{-\mu(\frac{t}{r}-\alpha)})^k (e^{-\mu(\frac{t}{r}-\alpha)})^{N-k}, \tag{18}$$

when $t \geq r\alpha$, and 0 otherwise.

V. Numerical Results and Discussion

We first verify the correctness of the expressions provided for the per iteration completion time statistics in (15) and (18) through Monte Carlo simulations generating 100000 independent realizations. Then, we show that MMC approach can reduce the average per-iteration completion time $E[T]$

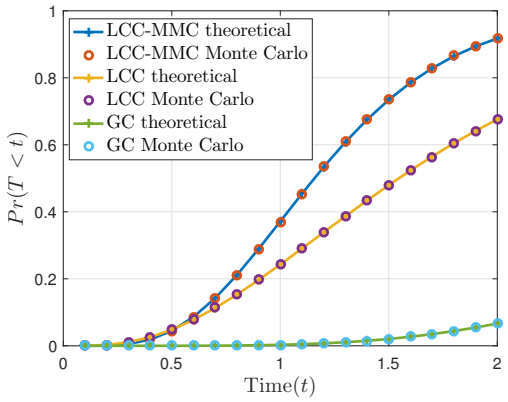


Fig. 1: Per iteration completion time statistics for $N = 10$ and $r = 5$.

significantly. In particular, we analyze the per iteration completion time of different DGD schemes, gradient coding (GC), Lagrange coded computation (LCC), and LCC with MMC (LCC-MMC). For the simulations we consider $M = N = 10$, $r = 5$, and use the cumulative density function (CDF) in (12) with parameters $\mu = 10$ and $\alpha = 0.01$ for the completion time statistics (we have made similar observations for a wide range of parameters).

In Fig.1 we plot the CDF of the per iteration completion time T for CG, LCC, and LCC-MMC schemes according to the closed-form expressions derived in Section 4 and Monte Carlo simulations. We observe from Fig. 1 that the provided closed-form expressions match perfectly with the results from the Monte Carlo simulations. We also observe that, LCC-MMC outperforms the LCC as well as GC scheme.

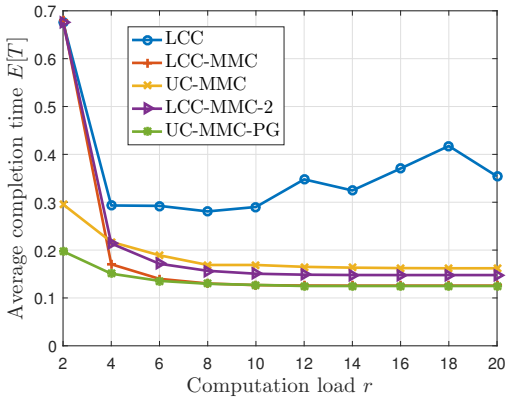
Next, we consider the setup from [4], where $M = 40$ workers are assigned $N = 40$ tasks to be computed at each iteration and analyze the performance of the various DGD schemes with respect to computation load r. Again, we use the distribution in (12) with parameters $\mu = 10$ and $\alpha = 0.01$. For the performance analysis, we consider both the average per iteration completion time $E[T]$ and the communication load, measured by the average total number of transmissions from the workers to the master, and the results obtained from 100000 Monte Carlo realizations are illustrated in Fig. 2. From Fig. 2(a), we observe that the UC-MMC scheme consistently outperforms LCC for all computation load values. More interestingly, UC-MMC performs very close to LCC-MMC, and for a small r, such as $r = 2$, it can even outperform UC-MMC. Hence, in terms of the computation load UC-MMC can be considered as a better option compared to LCC especially when r is low.

On the other hand, from Fig. 2(b) we observe that, in terms of the communication load the best scheme is LCC, while the UC-MMC introduces the highest communication load. We also observe that the communication load of LCC-MMC remains constant with r, whereas that of the LCC (UC-MMC) scheme monotonically decreases (increases) with r. Accordingly, the communication load of the LCC and UC-MMC schemes are closest at $r = 2$. From both Fig. 2(a) and Fig. 2(b) we note that, when r is low, e.g., when the workers have small storage capacity, UC-MMC may outperform the LCC scheme in terms of the average per iteration completion time including the decoding time as well.

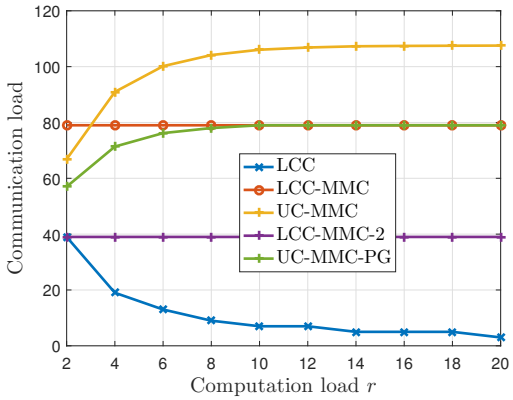
Remark 1. *An important aspect of the average per-iteration completion time that is ignored here, and by other works in the literature, is the decoding complexity at the master. Among these three schemes, UC-MMC has the lowest decoding complexity, while LCC-MMC has the highest. However, as discussed in Section II, the number of transmissions as well as the decoding complexity can be reduced via increasing the number of polynomials used in the decoding process. To illustrate this, we consider a different implementation of the LCC-MMC scheme, where two polynomials are used in the encoding part, denoted by LCC-MMC-2. In this scheme, for given r, the coded inputs correspond to the evaluation of two polynomials, each of degree $N-1$, at $r/2$ different points. Each worker sends a partial result to the master after execution of two computations, which correspond to the evaluation of these two polynomials at the same point. Since two polynomials are used in the encoding, the number of transmissions is reduced approximately to half compared to LCC-MMC as illustrated in Fig. 2(b). A noticeable improvement is achieved in the communication load, at the expense of a relatively small increase in the average per iteration completion time as illustrated in Fig. 2(a).*

Overall, the optimal strategy highly depends on the network structure. When the completion time is dominated by the workers' computation time, the LCC-MMC becomes the best alternative. This might be the case for special-purpose high performance computing (HPC) architectures employing message passing interface (MPI) rather than communicating through standard networking protocols [14]. On the other hand, if the communication load is the bottleneck, then LCC becomes more attractive especially when the workers have enough storage capacity, i.e., large r. However, as we observe in Fig. 2, the communication load and the average per iteration completion time can be balanced via playing with the number of polynomials used in the encoding process.

We also observe that when the workers have a small storage capacity, i.e., small r, UC-MMC has the lowest per iteration completion time. Moreover, when the decoding complexity is taken into account, UC-MMC can be preferable to coded computation schemes. Another advantage of the UC-MMC scheme is its applicability to K-batch strategy [15] where the parameter vector $\boldsymbol{\theta}_t$ is updated when any K gradient values, corresponding to different mini-batches (data points), are available at the master. Using gradients corresponding to K data points, instead of the full gradient, the per iteration completion time can be reduced. To this end, we consider a partial gradient scheme with MMC, called UC-MMC-PG, with 5% tolerance, i.e., $K = N \times 0.95$. The results in Fig. 2 show that when r is small, UC-MMC-PG can reduce the average completion time up to 70% compared to LCC, and up



(a) Average completion time vs. computation load.



(b) Communication load vs. computation load.

Fig. 2: Per iteration completion time and communication load statistics.

to 33% compared to UC-MMC; while only two gradient values are missing at each iteration. In addition to the improvement in average completion time, the UC-MMC-PG scheme can also reduce the communication load as shown in Fig. 2(b). We remark that, in the K-batch approach the gradient used for each update is less accurate compared to the full-gradient approach; however, since the parameter vector $\boldsymbol{\theta}_t$ is updated over many iterations, K-batch approach may converge to the optimal value faster than the full-gradient approach. Further, this tolerance rate can be dynamically updated through iterations to achieve better convergence results [16].

VI. Conclusions and Future Directions

We have introduced novel coded and uncoded DGD schemes when MMC is allowed from each worker at each iteration. We first provided a closed-form expression for the per iteration completion time statistics of these schemes under a shifted exponential computation time model, and verified our results with Monte Carlo simulations. Then, we compared these schemes with other DGD schemes in the literature in terms of the average computation and communication loads incurred.

We have observed that allowing multiple messages to be conveyed from each worker at each GD iteration can reduce the average completion time significantly by expoiting non-straggling workers at the expense of an increase in the average communication load. We also observed that UCUC with simple circular shift can be more efficient compared to coded computation approaches when the workers have limited storage capacity. We emphasize that, despite benefits of coded computation in reducing the computation time, their relevance in practical big data problems is questionable due to the need to jointly transform the whole dataset, which may not even be possible to store in a single worker.

References

[1] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1223–1231.

[2] 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, March 2018.

[3] R. K. Maity, A. S. Rawat, and A. Mazumdar, "Robust gradient descent via moment encoding with ldpc codes," *SysML Conference*, 2018.

[4] S. Li, S. M. M. Kalan, Q. Yu, M. Soltanolkotabi, and A. S. Avestimehr, "Polynomially coded regression: Optimal straggler mitigation via data encoding," *CoRR*, vol. abs/1805.09934, 2018.

[5] E. Ozfatura, S. Ulukus, and D. Gunduz, "Distributed gradient descent with coded partial gradient computations," *CoRR*, vol. 1811.09271, 2018.

[6] F. Haddadpour, Y. Yang, M. Chaudhari, V. R. Cadambe, and P. Grover, "Straggler-resilient and communication-efficient distributed iterative linear solver," *CoRR*, vol. abs/1806.06140, 2018.

[7] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. R. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *CoRR*, vol. abs/1801.10292, 2018.

[8] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 3368–3376.

[9] M. Ye and E. Abbe, "Communication-computation efficient gradient coding," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. StockholmsmÃd'ssan, Stockholm Sweden: PMLR, 10–15 Jul 2018, pp. 5610–5619.

[10] W. Halbawi, N. Azizan, F. Salehi, and B. Hassibi, "Improving distributed gradient descent using reed-solomon codes," in *2018 IEEE Int. Symp. on Inf. Theory (ISIT)*, June 2018, pp. 2027–2031.

[11] N. Ferdinand and S. C. Draper, "Hierarchical coded computation," in *2018 IEEE Int. Symp. Inf. Theory (ISIT)*, June 2018, pp. 1620–1624.

[12] A. Mallick, M. Chaudhari, and G. Joshi, "Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication," *CoRR*, vol. abs/1804.10331, 2018.

[13] N. Ferdinand, B. Gharachorloo, and S. C. Draper, "Anytime exploitation of stragglers in synchronous stochastic gradient descent," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Dec 2017, pp. 141–146.

[14] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CoRR*, vol. abs/1802.09941, 2018.

[15] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD," in *The 21st International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018.

[16] M. Teng and F. Wood, "Bayesian distributed stochastic gradient descent," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 6380–6390.