entropy

MDPI

Article

Straggler-Aware Distributed Learning: Communication–Computation Latency Trade-Off

Emre Ozfatura [1,*], Sennur Ulukus [2] and Deniz Gündüz [1]

[1] Information Processing and Communications Lab, Department of Electrical and Electronic Engineering, Imperial College London, London SW7 2AZ, UK; d.gunduz@imperial.ac.uk

[2] Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742, USA; ulukus@umd.edu

* Correspondence: m.ozfatura@imperial.ac.uk

Received: 10 April 2020; Accepted: 7 May 2020; Published: 13 May 2020

Abstract: When gradient descent (GD) is scaled to many parallel *workers* for large-scale machine learning applications, its per-iteration computation time is limited by *straggling* workers. Straggling workers can be tolerated by assigning redundant computations and/or coding across data and computations, but in most existing schemes, each non-straggling worker transmits one message per iteration to the parameter server (PS) after completing all its computations. Imposing such a limitation results in two drawbacks: *over-computation* due to inaccurate prediction of the straggling behavior, and *under-utilization* due to discarding partial computations carried out by stragglers. To overcome these drawbacks, we consider multi-message communication (MMC) by allowing multiple computations to be conveyed from each worker per iteration, and propose novel straggler avoidance techniques for both coded computation and coded communication with MMC. We analyze how the proposed designs can be employed efficiently to seek a balance between the computation and communication latency. Furthermore, we identify the advantages and disadvantages of these designs in different settings through extensive simulations, both model-based and real implementation on Amazon EC2 servers, and demonstrate that proposed schemes with MMC can help improve upon existing straggler avoidance schemes.

Keywords: coded computation; distributed computation; gradient descent; gradient coding; machine learning; parallel computing; polynomial codes

1. Introduction

Machine learning techniques have become highly popular thanks to their success in a wide variety of classification and regression tasks. This success can be partially attributed to the availability of high-quality large training datasets. Unfortunately, as the size of the datasets increases, memory storage, management and maintenance become unmanageable within the resources of a single machine. An efficient way to deal with such colossal computing tasks within a reasonable training time is to exploit computation and memory resources of multiple machines in parallel.

In many supervised machine learning problems, the objective is to minimize the following *parameterized empirical loss function* for a given training dataset $\mathcal{D}$ of $(\mathbf{x}, y)$ pairs, where $\mathbf{x}$ denotes the input sample while y is the output (label for classification problems):

$$L(\boldsymbol{\theta}) \triangleq \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},y)\in\mathcal{D}} l\left((\mathbf{x}, y), \boldsymbol{\theta}\right), \tag{1}$$

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

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

where η_t is the learning rate at iteration t, and the gradient with respect to current parameter vector is given by

$$\nabla_{\boldsymbol{\theta}_t} L(\boldsymbol{\theta}_t) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},y)\in\mathcal{D}} \nabla_{\boldsymbol{\theta}_t} l\left((\mathbf{x}, y), \boldsymbol{\theta}_t\right). \tag{3}$$

When the dataset $\mathcal{D}$ is large, distributed GD (DGD) [1–4] can be used to reduce the computation time, and hence, the overall training time. In the naive *parameter server (PS)* type implementation with K workers, denoted by $w_1, \ldots w_K$, $\mathcal{D}$ is divided into K non-overlapping equal-size subsets $D_1, \ldots, D_K$, where each subset is assigned to a different worker. At the beginning of iteration t, the PS broadcasts the current parameter vector $\boldsymbol{\theta}_t$ to all the workers. Worker w_k computes the partial gradient $\mathbf{g}_k^{(t)}$ with respect to $\boldsymbol{\theta}_t$, based on the assigned dataset, i.e.,

$$\mathbf{g}_k^{(t)} = \frac{1}{|\mathcal{D}_k|} \sum_{(\mathbf{x},y)\in\mathcal{D}_k} \nabla_{\boldsymbol{\theta}_t} l((\mathbf{x}, y), \boldsymbol{\theta}_t). \tag{4}$$

The PS waits until it receives the partial gradients, $\mathbf{g}_k^{(t)}$, from all the workers, aggregates them to obtain the full gradient $\nabla_{\boldsymbol{\theta}_t} L(\boldsymbol{\theta}_t) = \frac{1}{K}\sum_{k=1}^{K} \mathbf{g}_k^{(t)}$, and updates the parameter vector according to (2). In this implementation, due to *synchronized updates* the completion time of each iteration is constrained by the slowest, so-called *straggling worker(s)*, which can be detrimental for the convergence of the algorithm. Please note that although the straggling behavior is often used to describe the processing delay of a worker, we use this term for a wide range of delays including connection failures or congestion delays, which are likely to increase with the number of workers involved.

A wealth of straggler avoidance techniques have been proposed in recent years for DGD as well as other distributed computation tasks [5–48]. The common design notion behind all these schemes is the assignment of redundant computations/tasks to workers, such that faster workers can compensate for the stragglers. The main challenge is that the computation and communication latency of workers vary over time, and these values are not known in advance. This randomness can be treated as packet erasures in a communication channel [17], and erasure coding techniques can be used to efficiently combat stragglers rather than simple task replication [12–15,19–21,30]. However, most of the existing techniques, such as gradient coding (GC) [12], Lagrange coded computation (LCC) [20], and their variations, suffer from two drawbacks: *over-computation* and *under-utilization*. By assigning redundant computations to workers, each iteration can be terminated with results from only a subset of the workers, and the minimum number of workers that must complete the assigned computation is called the *non-straggling threshold*. The non-straggling threshold can be reduced by increasing the redundancy; however, a smaller threshold does not necessarily imply a lower completion time. Workers may be assigned more tasks than required due to an inaccurate prediction of the straggling behavior, which we refer to as *over-computation*. Moreover, in those schemes straggling behavior is treated as 'all or nothing' (straggler/non-straggler), and the computations carried out by stragglers are discarded as long as they cannot complete all their assigned computations. However, in practice, *non-persistent* straggling servers can complete a certain (sometimes significant) portion of their assigned tasks. This leads to *under-utilization* of the computational resources. Therefore, our main objective in this paper is to introduce straggling avoidance techniques that can mitigate under-utilization and over-computation. This will be achieved by allowing each worker to send multiple messages to the PS at each iteration, which we refer to as *multi-message communication (MMC)*. However, MMC may introduce additional delays due to the communication overhead. Hence, in this paper we also address

the communication–computation latency trade-off, and provide flexible designs that can balance the two.

Our contributions can be summarized as follows. First, we propose new straggler avoidance techniques designed to benefit from MMC. Second, to account for the additional communication load that may be introduced due to MMC, we provide designs that can provide a balance between the communication and computation latencies. Third, through extensive numerical simulations we illustrate the main advantages/disadvantages of the proposed designs compared to the existing ones. Finally, based on real experiments on Amazon EC2 servers, we show that the proposed schemes can improve upon existing straggler avoidance techniques.

2. An Overview of Existing Straggler Avoidance Techniques

There is already a rich literature on straggler avoidance methods in distributed learning/computation, many of them employing some form of coding. To provide a better understanding, we classify those schemes under three groups based on whether coding is employed or not, and if so, at which stage; namely (1) coded computation, (2) coded communication, and finally, (3) uncoded computation. Classification of the existing straggler-aware DGD schemes in the literature is given in Table 1. Before explaining these schemes, we first introduce two design parameters: *computation load* and *communication load*. Computation load, denoted by r, measures the redundancy of computations assigned to each worker compared to naive distributed computation, where each computation task is assigned to a single worker. Communication load characterizes the total number of messages conveyed from the workers to the PS per iteration, where the size of each message is equal to the size of the parameter vector, d.

Table 1. Classification of the DGD algorithms in the literature according to the straggler avoidance approach used.

Uncoded Computation	Coded Transmission	Coded Computation
[5–11,49]	[12–16]	[17–34]

2.1. Coded Computation Schemes

In some problems, the gradient can be expressed as an explicit function of the dataset and the parameter vector, and more efficient straggler mitigation techniques can be introduced exploiting this particular relation. For example, for the least-squares linear regression problem, the loss function can be explicitly written as

$$L(\boldsymbol{\theta}) = \frac{1}{2N} \sum_{(\mathbf{x},y)\in\mathcal{D}} (y - \mathbf{x}^T\boldsymbol{\theta})^2 \,, \tag{5}$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input vector, $y \in \mathbb{R}$ the corresponding output, and N is the size of the dataset. For this particular loss function, the gradient is given by

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}, \tag{6}$$

where $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]^T$ and $\mathbf{y} = [y_1, \ldots, y_N]^T$ are concatenation of all input vectors and output values, respectively. Since the second term, $\mathbf{X}^T\mathbf{y}$, does not include the term $\boldsymbol{\theta}$, it remains the same throughout the iterations. Therefore, the main computation task is to compute $\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}_t$ at each iteration. In this particular case the problem can be reduced to distributed matrix-matrix multiplication, or matrix-vector multiplication if $\mathbf{X}^T\mathbf{X}$ is computed beforehand, and this simplified form allows exploiting novel ideas from coding theory.

In the naive distributed computation scenario, $\mathbf{X}$ can be divided into K submatrices (assume,

for simplicity, that K divides N), $\mathbf{X}_1, \ldots, \mathbf{X}_K$, each of size $N/K \times d$, such that kth worker computes $\mathbf{X}_k^T \mathbf{X}_k \boldsymbol{\theta}_t$ at iteration t. Since the following equality holds

$$\mathbf{X}^T \mathbf{X} \boldsymbol{\theta} = \sum_{k=1}^{K} \mathbf{X}_k^T \mathbf{X}_k \boldsymbol{\theta}, \tag{7}$$

PS can obtain the full gradient receiving the computation results from all the workers. In contrast to the naive approach, coded computation schemes for distributed matrix multiplication [22,23,32,34] first encode the submatrices, and then assign them to the workers to achieve a certain tolerance against slow/straggling workers.

We note that $\mathbf{W} \triangleq \mathbf{X}^T \mathbf{X}$ in (6) also remains unchanged throughout GD iterations. Hence, if $\mathbf{W}$ can be computed at the beginning, the main computational task reduces to linear operations at each iteration, which allows employing various linear coding structures, e.g., maximum distance separable (MDS) codes, or rateless codes, to encode rows of $\mathbf{W}$ to achieve robustness against stragglers [17–19,25,28,29].

We want to reemphasize that coded computation schemes are mostly designed for the full recovery of the main task, such as the recovery of the full gradient in DGD. However, in practice, approximate/partial gradients are commonly used instead of the full gradient to seek a balance between the computation time and accuracy, and to eventually reduce the convergence time. Approximate GC and partial gradient recovery schemes have also been studied in [50–52] and [29,53,54], respectively. In the scope of this paper, we limit our focus to full-gradient recovery and leave the MMC variation of partial gradient recovery [29] as a future work.

2.2. Coded Transmission Schemes

Let $\mathcal{G} = \{\mathbf{g}_1, \ldots, \mathbf{g}_K\}$ be the set of partial gradients corresponding to datasets $\mathcal{D}_1, \ldots, \mathcal{D}_K$. In the GC scheme with computation load r, r partial gradient computations, denoted by $\mathcal{G}_k$, are assigned to worker k [12]. After computing these r partial gradients, each worker sends a linear combination of the results,

$$\mathbf{c}_k^{(t)} \triangleq \mathcal{L}_k(\mathbf{g}_i^{(t)} : \mathbf{g}_i \in \mathcal{G}_k). \tag{8}$$

We refer to these linear combinations $\mathbf{c}_1, \ldots, \mathbf{c}_K$ as *coded partial gradients*. The PS waits until it receives sufficiently many coded partial gradients to recover the full gradient. It is shown in [12] that for any set of non-straggler workers $\mathcal{W} \subseteq [K]$ with $|\mathcal{W}| = K - r + 1$, there exists a set of coefficients $\mathcal{A}_\mathcal{W} = \{a_k : k \in \mathcal{W}\}$ such that

$$\sum_{k \in \mathcal{W}} a_k \mathbf{c}_k^{(t)} = \frac{1}{K} \sum_{k=1}^{K} \mathbf{g}_k^{(t)}. \tag{9}$$

Hence, GC can tolerate up to $r - 1$ persistent stragglers at each iteration. GC can also be interpreted as a polynomial interpolation problem [14]. In this model, the gradient assignment matrix is called a *mask matrix*, and the *support* of the kth row $\mathbf{M}_{(k,:)}$, denoted by $supp(\mathbf{M}_{(k,:)})$, gives the index of the partial gradients assigned to w_k, $\mathcal{G}_k$, and for given redundancy r, $||\mathbf{M}_{(k,:)}||_1 = r$. For a given mask matrix $\mathbf{M}$, GC is equivalent to interpolating a polynomial with *degree* h, where $h = K - \min_k ||\mathbf{M}_{(:,k)}||_1$; in other words, h is equal to the number of zeros in the most sparse column of $\mathbf{M}$. We remark that if $\mathbf{M}$ is a $K \times K$ matrix then $||\mathbf{M}_{(:,k)}||_1 = ||\mathbf{M}_{(k,:)}||_1 = r$, $\forall k \in [K]$.

In a broad sense, each partial gradient $\mathbf{g}_k$ is embedded into a polynomial f_k, and each worker evaluates the polynomials $f_1, \ldots, f_K$ at preassigned points, and sends their sum to the PS. Let polynomial f_k be constructed as

$$f_k(x) = \prod_{i:\mathbf{g}_k \notin \mathcal{G}_i} (x - \alpha_i) \tag{10}$$

for some distinct $\alpha_1, \ldots, \alpha_K$. We define another polynomial:

$$h(x) = \sum_{k=1}^{K} \mathbf{g}_k f_k(x). \tag{11}$$

At each iteration, each worker w_i sends $h(\alpha_i)$ to the PS. The key design trick here is that worker w_i does not need to compute $\mathbf{g}_k$, if $f_k(x)$ has a root at α_i, and can compute $h(\alpha_i)$ only with the knowledge of $\mathbf{g}_k$s in the set $\mathcal{G}_i$.

To explain the decoding stage, consider the following mask matrix:

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{12}$$

There will be six partial gradients $\mathbf{g}_1, \ldots, \mathbf{g}_6$ and six corresponding polynomials $f_1(x), \ldots, f_6(x)$ to embed their values. Observe that the number of zeros in each column, $K - r$, is equivalent to the number of roots of the corresponding polynomial f, which is three for all polynomials, in our example. Then, the leading coefficient of $h(x)$ is equal to $\mathbf{g} = \sum_{k=1}^{6} \mathbf{g}_k$, and it has degree three since the degree of each polynomial f_k is three. Therefore, at each iteration, $h(x)$ can be interpolated using its value at any $K - r + 1$, 4 for the given example, different points. Accordingly, to recover $\mathbf{g}$, any $K - r + 1$ results are sufficient, which implies a non-straggling threshold of $K_{GC}(r) = K - r + 1$. We sketch the general design strategy and the corresponding non-straggling threshold; however implementation of the encoding and decoding procedures, and their complexity (see [14] for further details), also affect the completion time; nevertheless, in the scope of this paper we omit the complexity analysis and focus on computation and communication latency.

In [13], the GC scheme is extended to seek a trade-off between the communication latency and the non-straggler threshold, and it is shown that the length of the coded partial gradient $\mathbf{c}_k$ can be reduced with an increase in the non-straggling threshold. The trade-off between the communication latency and straggler tolerance is also studied in [15], and it is shown that the PS can recover the full gradient faster when each worker is allowed to send more than one coded partial gradient. We classify these schemes as coded transmission since computations are carried out using uncoded data, but the computations are transmitted to the PS in a coded manner.

2.3. Uncoded Computation Schemes

This class includes schemes that do not employ any coding. In the naive distributed approach, the computation task is divided into disjoint subtasks to be executed in parallel. To mitigate the stragglers each worker may perform some backup computations [5–7,21], certain unfinished subtasks (slow workers) can be relaunched at the fast workers [10,11], or some additional backup workers can be employed [9]. Alternatively, PS can terminate an iteration after receiving results from a subset of workers [49,55].

Existing straggler tolerant DGD schemes focus on minimizing the non-straggling threshold, which does not necessarily capture the average completion time statistics for one iteration of the GD algorithm. Indeed, in certain regimes of computation load r, the average completion time may be increasing as the non-straggling threshold decreases. Accordingly, in this paper, we consider the average completion time as the main performance metric, and allow workers to send multiple messages at each iteration to reduce the per-iteration completion time.

MMC can be easily applied in uncoded computation by assigning each computation task to multiple workers [7,21]. Workers can then return each of their computations as soon as it is completed,

and the iteration is completed when each computation task is completed by at least one worker. Multi-message coded computation is also studied in [18,25]. However, these schemes are limited to matrix-vector multiplication. Furthermore, they ignore the communication overhead due to MMC and its impact on the communication latency, and focus only on the computation time.

3. Coded Computation with MMC

For the coded computation we employ the LCC method introduced in [20,22], which uses polynomial interpolation for the code design. In this section, we first explain the structure of the Lagrange polynomial, then explain how it is used for coded computation, and finally discuss how it can be modified to benefit from MMC.

3.1. Lagrange Coded Computation (LCC)

First, $\mathbf{X}$ is divided into K submatrices (assume, for simplicity, that K divides N), $\mathbf{X}_1, \ldots, \mathbf{X}_K$, each of size $N/K \times d$. For given r, assuming K is divisible by r, these K submatrices are further divided into r disjoint groups, each containing K/r submatrices. Let $\mathbf{X}_{q,i}$ denote the ith submatrix in the qth group, and $\mathbf{X}_q$ denote all the submatrices in the qth group; that is, $\mathbf{X}_q$ is an $N/r \times d$ submatrix of $\mathbf{X}$. Then, for distinct real numbers $\alpha_1, \ldots, \alpha_{K/r}$, we form the following r structurally identical polynomials of degree $K/r - 1$, taking the submatrices of $\mathbf{X}_q$ as their coefficients:

$$f_q(z) = \sum_{i=1}^{K/r} \mathbf{X}_{q,i} \prod_{j=1, j\neq i}^{K/r} \frac{z - \alpha_j}{\alpha_i - \alpha_j}, \; q \in [r], \tag{13}$$

which satisfy $f_q(\alpha_i) = \mathbf{X}_{q,i}$, $\forall k, i$. Then, we define

$$H(z) \triangleq \sum_{q=1}^{r} f_q(z)^T f_q(z) \boldsymbol{\theta}_t. \tag{14}$$

Coded submatrices $\tilde{\mathbf{X}}_k^{(q)}$, $q \in [r]$, for worker w_k, $k \in [K]$ are obtained by evaluating $f_q(z)$ polynomials at distinct values, $\beta_k \in \mathbb{R}$, i.e., $\tilde{\mathbf{X}}_k^{(q)} = f_q(\beta_k)$. At each iteration w_k returns the value of

$$H(\beta_k) = \sum_{q=1}^{r} (\tilde{\mathbf{X}}_k^{(q)})^T \tilde{\mathbf{X}}_k^{(q)} \boldsymbol{\theta}_t. \tag{15}$$

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

$$\sum_{i=1}^{K/r} H(\alpha_i) = \mathbf{X}^T \mathbf{X} \boldsymbol{\theta}_t, \tag{16}$$

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 $K_{LCC}(r) = 2\lceil N/r \rceil - 1$.

3.2. LCC with MMC

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

$$f(z) \triangleq \sum_{i=1}^{K} \mathbf{X}_i \prod_{j=1, j\neq i}^{K} \frac{z - \alpha_j}{\alpha_i - \alpha_j}, \tag{17}$$

where $\alpha_1, \ldots, \alpha_K$ are K distinct real numbers, and we construct

$$h(z) \triangleq f(z)^T f(z)\boldsymbol{\theta}_t, \tag{18}$$

such that $h(\alpha_i) = \mathbf{X}_i^T \mathbf{X}_i \boldsymbol{\theta}_t$. Consequently, if the polynomial $h(z)$ is known at the PS, then the full gradient $\sum_{i=1}^{K} h(\alpha_i) = \sum_{i=1}^{N} \mathbf{X}_i^T \mathbf{X}_i \boldsymbol{\theta}_t$ can be obtained. To this end, Kr coded submatrices $\tilde{\mathbf{X}}_k^{(q)}, k \in [K], q \in [r]$, are constructed by evaluating $f(z)$ at Kr different points, $\beta_k^{(q)}$, i.e.,

$$\tilde{\mathbf{X}}_k^{(q)} = f(\beta_k^{(q)}), \ k \in [K], q \in [r], \tag{19}$$

and $\tilde{\mathbf{X}}_k^{(1)}, \tilde{\mathbf{X}}_k^{(r)}$ are assigned to w_k, $k \in [K]$. w_k computes $(\tilde{\mathbf{X}}_k^{(1)})^T \tilde{\mathbf{X}}_k^{(1)} \boldsymbol{\theta}_t, \ldots, (\tilde{\mathbf{X}}_k^{(r)})^T \tilde{\mathbf{X}}_k^{(r)} \boldsymbol{\theta}_t$ sequentially, and transmits each of these results to the PS as soon as it is computed. Coded computation corresponding to coded data point $\tilde{\mathbf{X}}_k^{(q)}$ at w_k provides the value of polynomial $h(z)$ at point $\beta_k^{(q)}$. The degrees of polynomials $f(z)$ and $h(z)$ are $K-1$ and $2(K-1)$, respectively, which implies that $h(z)$ can be interpolated from its values at any $2K-1$ distinct points. Hence, any $2K-1$ 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 used; 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 explored in Section 7.

4. GC with MMC

In the original GC scheme of [12], the number of messages transmitted to the PS per-iteration per-worker is limited to one. Due to the synchronized model update, the workers that complete their computations stay idle until they receive the updated parameter vector to start the next iteration. To prevent under-utilization of the computation resources, we will allow each worker to send coded partial gradients to the PS; that is, at each iteration each worker sends multiple coded partial gradients instead of sending a single coded computation result. In the scope of this paper, we will present two different approaches to design coded partial gradients, namely *correlated code design* and *uncorrelated code design*, which are explained next.

4.1. Correlated Code Design

In GC, the number of partial gradients that are linearly combined to form the transmitted message from a worker is equal to the computation load, r. In MMC, we allow each worker to compute and transmit multiple coded partial gradients, each of which will be generated by combining $m \leq r$ gradient computations. We will refer to m as the *order* of the corresponding partial gradient. In particular, each worker will be able to send up to $l = r - m + 1$ different messages, each of order m; that is, each of the coded partial gradients will be a linear combination of the m most recently computed partial gradients.

The proposed scheme consists of two phases: the assignment phase and the computation phase. In the assignment phase, executed only at the beginning of the training process, partial gradient computations are assigned to workers according to matrix $\mathbf{M}$, which is constructed using cyclic shifts as in the GC scheme. Then, in the computation phase, repeated at each iteration, each user computes the assigned partial gradients based on the given order, and as soon as it finishes computing the first m of them, it starts sending the coded partial gradients to the PS. We remark that the encoding function $\mathcal{L}$ used to construct coded partial gradients is identical for all the workers; thus, the coded partial gradient depends only on the partial gradients and their order. Furthermore, the encoding function

$\mathcal{L}$ is the one used for the GC scheme with $r = m$. The overall procedure is illustrated in Algorithm 1. Next, we provide an example to clarify the proposed strategy.

Algorithm 1 GC with MMC (correlated design)

Assignment phase:
For $k \in [K]$, construct $\mathcal{G}_k$ based on $\mathbf{M}$
Computation phase:
for user $k \in [K]$ **do** in parallel
  for $j = 1, \ldots, r$ **do**
    Compute assigned jth gradient $\mathcal{G}_k(j)$
    if $j \geq m$ **then**
      Send $\mathbf{c} = \mathcal{L}(\mathcal{G}_k(j), \ldots, \mathcal{G}_k(j-(m-1))$

Example 1. *Let $K = 6$, $r = 3$, $m = 2$, and consider the assignment matrix $\mathbf{M}$, whose ith row indicates the mini-batches assigned to the ith worker; that is $\mathbf{M}(i, j) = 1$ means that partial gradient $\mathbf{g}_j$ will be computed by the ith worker. In the rest of the paper, to simplify the notation we drop the time index from the gradients when we focus on a single iteration of the algorithm. In GC with $K = 6$ and $r = 3$, we have the following assignment matrix.*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{20}$$

When $m = 2$, coded gradients are obtained according to the assignment matrix $\tilde{M}$, which is obtained by removing the last $r - m$ of the 1s in each row (shown in red above). When the assignment matrix $\tilde{M}$ is used to design GC, a total of $K = 6$ coded partial gradients, each of order two, are constructed; and the full gradient can be obtained from any $K - m + 1 = 5$ coded partial gradients. Let $\mathbf{c}_1, \ldots, \mathbf{c}_6$ denote the corresponding coded partial gradients. We remark that $\mathbf{c}_1$ is a linear combination of $\mathbf{g}_1$ and $\mathbf{g}_2$, while $\mathbf{c}_2$ is a linear combination of $\mathbf{g}_2$ and $\mathbf{g}_3$. Since $\mathbf{g}_1, \mathbf{g}_2$ and $\mathbf{g}_3$ are assigned to the first worker, it can send both coded messages $\mathbf{c}_1$ and $\mathbf{c}_2$. To illustrate the assignment of coded partial gradients, we use the assignment matrix $\mathbf{C}$, where the ith column shows the assigned coded gradients to the ith worker in the order of computation. In Example 1, we have

$$\mathbf{C} = \begin{bmatrix} \mathbf{c}_1 & \mathbf{c}_2 & \mathbf{c}_3 & \mathbf{c}_4 & \mathbf{c}_5 & \mathbf{c}_6 \\ \mathbf{c}_2 & \mathbf{c}_3 & \mathbf{c}_4 & \mathbf{c}_5 & \mathbf{c}_6 & \mathbf{c}_1 \end{bmatrix}. \tag{21}$$

We call this approach correlated code design, *since the same coded partial gradient can be computed and sent by more than one worker, e.g., in Example 1, $\mathbf{c}_2$ can be sent by both w_1 and w_2. In Example 1, the original GC algorithm needs to receive computations from at least four workers to complete an iteration; whereas the proposed scheme can complete an iteration with results from only three workers. For instance, when workers 1, 2 and 4 each send two coded partial gradients, the PS will obtain $\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4, \mathbf{c}_5$, and recover the full gradient. In the next section, we will analyze the uncorrelated coded design approach, where each coded gradient is assigned to exactly one worker.*

4.2. Uncorrelated Code Design

Here, we present yet another code construction to extend GC to the MMC scenario. Unlike the previous design, now, we allow each coded partial gradient to have a different order and also to use a different encoding strategy, so that a particular coded partial gradient is sent by at most one user. Since the PS does not receive a coded partial gradient from more than one user, we call this scheme

uncorrelated code design. Uncorrelated code design for GC with MMC is defined by the order vector $[m_0, \ldots, m_{l-1}]$, where $r > m_i$ is the order of the $(i+1)$th coded partial gradient and the order vector also defines when to send a coded partial gradient; that is, each worker sends the $(i+1)$th coded partial gradient when it computes the first m_i assigned partial gradients. The overall procedure is illustrated in Algorithm 2.

Algorithm 2 GC with MMC (uncorrelated design)

Assignment phase:
For $k \in [K]$, construct $\mathcal{G}_k$ based on $\mathbf{M}$
Computation phase:
for user $k \in [K]$ **do** in parallel
    Initialize $i = 0$
    for $j = 1, \ldots, r$ **do**
        Compute assigned jth gradient $\mathcal{G}_k(j)$
        if $j = m_i$ **then**
            Send $\mathbf{c} = \mathcal{L}(\mathcal{G}_k(j), \ldots, \mathcal{G}_k(1))$
            $i = i + 1$

Before presenting the main result, we give an example to clarify the procedure. In the example, the partial gradient assignments to 6 workers is governed by the mask matrix in (12). Assume that each worker sends a coded partial gradient after computing the first two assigned partial gradients, and then sends a second coded partial gradient after computing all its assigned partial gradients. Now, consider the scenario with 12 workers and the following mask matrix:

$$\tilde{\mathbf{M}} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{22}$$

According to $\tilde{\mathbf{M}}$ three partial gradients are assigned to six workers, whose rows are shown in red, while two partial gradients are assigned to the remaining six workers, whose rows are shown in blue. In terms of encoding/decoding process these two are equivalent. Therefore, sending an additional coded partial gradient corresponds to adding a "virtual" worker, i.e., the rows in red correspond to the real workers, while the rows in blue to the virtual ones. We note that given $\tilde{\mathbf{M}}$, degree of $h(x)$ and the non-straggling threshold will be 7 and 8, respectively, since there are exactly seven zeros in each column.

We are reminded that in the original GC scheme, the PS waits for 4 workers to recover the full gradient, while in the proposed scheme each worker can send a coded partial gradient as a virtual worker, after only two computations, and the full gradient can be recovered from any 8 coded partial gradients, including those from the virtual workers. Assume, for example, that three of the workers are non-stragglers, and each of them sends 2 coded partial gradients, while two workers are non-persistent stragglers, and each of them sends only one coded gradient, while the last worker is a persistent

straggler. In this case, the full gradient can be obtained by the proposed approach but not with the original GC scheme. Hence, the proposed approach improves the per-iteration completion time. In general, the following lemma highlight the recovery performance of the uncorrelated design.

Lemma 1. *For given K and order vector $[m_0, \dots, m_{l-1}]$ of length l, any $Kl - (\sum_{i=0}^{l-1} m_i) + 1$ coded partial gradients are sufficient to recover full gradient.*

The proof of Lemma 1 follows from the polynomial interpolation strategy explained in Section 2.2, where each gradient is embedded into a polynomial. Since each worker is allowed to send l messages per iteration, we can introduce $K(l-1)$ "virtual" workers such that m_i partial gradients are assigned to the ith virtual worker, resulting in a total of Kl workers. Then, we design a GC scheme based on the mask matrix of these Kl workers. Given the order vector $\mathbf{m} = [m_0, \dots, m_{l-1}]$, the number of zeros in any column of the mask matrix M is given by $Kl - (\sum_{i=0}^{l-1} m_i)$; and thus, $Kl - (\sum_{i=0}^{l-1} m_i) + 1$ coded partial gradients are required to recover the full gradient.

We note that while the use of coded partial gradients with lower orders increases the recovery threshold, they can be obtained faster, as they allow the PS to exploit the computations carried out by non-persistent stragglers. We leave the optimization of the partial gradient orders depending on system parameters and requirements as future work.

Another important issue regarding MMC is the *communication load*, which denotes the average number of messages received by the PS at each iteration. The communication load increases with the number of virtual workers; therefore, the optimal MMC strategy depends critically on the communication architecture of the network and the protocol used to transmit messages from the workers to the PS as well as the computation speeds of the workers.

4.3. Clustering

Next, we introduce *clustering*, which can further speed up the average iteration time. We divide the workers into P equal-size disjoint clusters, where the set of workers in cluster p is denoted by $\mathcal{W}_p \subset \mathcal{W}$, $p \in [P]$. Dataset and the corresponding set of partial gradients $\mathcal{G} = \{\mathbf{g}_1, \dots, \mathbf{g}_K\}$ are also divided into P equal-size disjoint subsets, and the set of partial gradients assigned to the pth cluster is denoted by $\mathcal{G}_p$. In the clustering approach, the workers in the pth cluster are responsible for computing

$$\frac{1}{|\mathcal{G}_p|} \sum_{k \in \mathcal{G}_p} \mathbf{g}_k, \tag{23}$$

and the GC scheme is applied to each cluster independently. The advantage of the clustering approach is that when GC is applied with clustering, it is possible to tolerate $r-1$ stragglers in each cluster; hence, in the best scenario, which is when the stragglers are uniformly distributed among the clusters, it is possible to tolerate $p(r-1)$ stragglers in total. On the other hand, in the worst-case scenario; that is, when the stragglers are accumulated in a particular cluster, it is possible to tolerate only $r-1$ stragglers, which is equivalent to the performance of the GC scheme. Therefore, for a particular straggler realization, if the full gradient can be recovered with GC (without clustering), then it can also be recovered with clustering, while the converse is not true. This implies that it is possible to achieve a lower iteration time on average when clustering is employed.

To illustrate the above, we consider the case with $K = 10$ and $r = 3$. When GC is applied, 8 non-straggler workers are required to recover the full gradient. Alternatively, clustering the workers into $P = 2$ clusters, 3 non-straggler workers from each cluster are required for full-gradient recovery. Any straggler realization that is "good" for GC (i.e., not more than 2 stragglers) is also good for clustering; however there are certain realizations that are good for the latter, but not for the former. To illustrate this, in Figure 1, we depict two different straggler realizations. One can observe that Realization 1 is good for both schemes, while in Realization 2, the full-gradient recovery can be achieved by only the clustering strategy. We want to emphasize that although 6 non-straggling workers in

Realization 1 are sufficient for full-gradient recovery, this does not mean that any 6 non-straggling workers would be sufficient. However, the set of straggler realizations where the full-gradient recovery is possible using GC is a subset of the one for clustering. Consequently, while the non-straggling threshold is the same for GC and GC with clustering, this threshold only represents the worst-case scenario and as exemplified above, the probability of reaching recovery condition is higher when clustering is employed; and hence the average computation time can be reduced.

At this point, we remark that the fractional repetition scheme in [12] is a special case of the proposed clustering approach, where the size of a cluster is equal to the computation load, r. As an example consider $K = 40$, $r = 10$ and $P = 4$, where the workers are divided into $P = 4$ clusters, while the mini-batches are divided into 4 subsets, and each cluster is responsible for a different subset. In the fractional repetition scheme, the PS waits until at least one worker from each cluster completes and sends its partial gradient. One can observe that if at least $K - r + 1 = 31$ workers complete and send their computations to the PS, there must be at least one worker from each cluster; hence, the non-straggling threshold is $K - r + 1$. However, the non-straggling threshold represents a worst-case scenario. Notice that even 4 workers, each from a different cluster can be sufficient to obtain the full gradient. On the other hand, the cyclic repetition scheme in [12], which has a circulant mask matrix as in (12), always has to wait until receiving coded messages from at least $K - r + 1$ workers. Although both GC schemes achieve the same optimal non-straggling threshold, their average performance may differ substantially. Fractional repetition scheme requires K to be an integer multiple of r, whereas the clustering approach outlined above can be applied to any (K, r) pair.

We note that when MMC is allowed, clustering may also be disadvantageous. On one hand, more straggling workers can be tolerated on average, on the other hand GC is applied to each cluster independently; hence, a coded partial gradient from a particular cluster cannot be used for another cluster. Consequently, the optimal clustering strategy with MMC depends on the computation statistics of the workers.

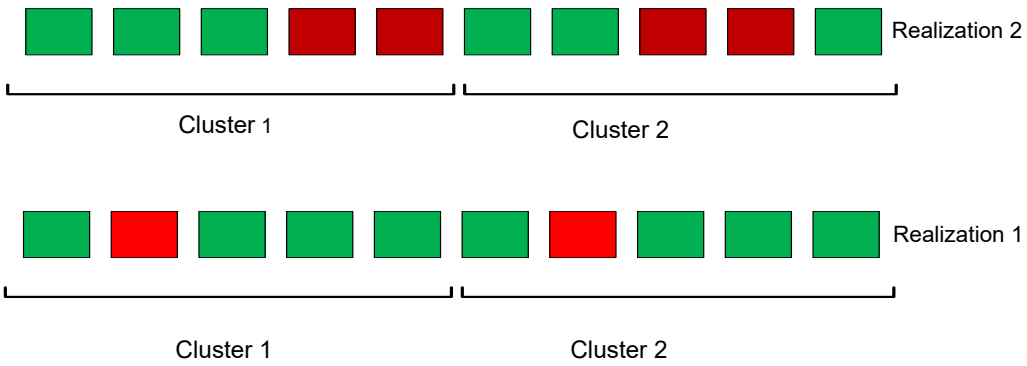


Figure 1. Two possible straggler realizations where green and red blocks illustrate the straggler and non-straggler workers, respectively.

4.4. Hybrid Implementation

The optimal DGD strategy depends critically on the computation time statistics of the workers. In particular, when the computation speeds of the workers are similar, MMC is expected to have a better performance as it can exploit all the computations carried out across the workers; however, when one of the workers is much faster compared to the others, fractional repetition can be preferred. To illustrate this trade-off, consider the case $K = 5$ and $r = 5$. With the fractional repetition scheme, the PS waits for the fastest worker to finish all the assigned computations; however, with GC with MMC for given order vector $\mathbf{m} = [5, 3]$, the PS waits for 3 coded messages sent from 2 workers; hence the overall speed will depend on the second, or even the third fastest worker.

Accordingly, we can propose a hybrid scheme, in which the workers initially behave as dictated by the GC-MM scheme, but if a worker is fast enough to complete all its computations, then it switches to fractional repetition scheme, and sends the average gradient instead of a coded partial gradient.

5. Uncoded Computation with MMC

In uncoded computation, dataset $\mathcal{D}$ is divided into K non-overlapping equal-size subsets $D_1, \ldots, D_K$, where $\mathbf{g}_k$ denotes the partial gradient corresponding to dataset $\mathcal{D}_K, k \in [K]$. To tolerate straggling workers more than one partial gradient is assigned to each worker according to a certain order. Hence, uncoded computation is defined by a partial gradient assignment and order of computation. Let $\mathbf{M}$ be the assignment matrix for the partial gradients to workers, where $\mathbf{M}(j,i) = k$ means that the kth partial gradient $\mathbf{g}_k$ is computed by the ith worker in the jth order. This assignment can be random [5], or according to a certain structure [7,29]. In this paper, we consider the circular shifted assignment strategy, similar to the one used for GC:

$$\mathbf{M}(j,:) = \text{circshift}\,([1:N], -(j-1)). \tag{24}$$

For instance, for $K = 10$ and $r = 4$, we have:

$$\mathbf{M} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 1 \\ 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 1 & 2 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 1 & 2 & 3 \end{bmatrix}.$$

We highlight that uncoded computation is actually a special case of the GC with MMC scheme, with message order $m = 1$. We remark that the necessary condition to obtain the full gradient, with GC and its multi-message variations, is that each partial gradient is computed by at least one worker. It is easy to see that uncoded computation will always outperform GC if we only consider the computation time. Therefore, the main advantage of the GC scheme is to reduce the communication overhead.

Although we limit our focus to full-gradient recovery in this paper, a partial gradient can be also used to update the parameter vector at each iteration [49]. We will show in Section 7 that significant gains can be obtained in both computation time and communication load by ignoring only 5% of the partial gradients. Lastly, we note that under the assumption of independent and identically distributed (i.i.d.) delays over time and over workers, the obtained partial gradient will be an *unbiased estimate* of the full gradient as in the stochastic gradient descent (SGD) approach.

6. 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 K workers, and assume that the dataset is also divided into K subsets. For the straggling behavior, we adopt the model in [17] and [18], and assume that the probability of completing s computations at any server, performing s identical matrix-vector multiplications, by time t is given by

$$F_s(t) \triangleq \begin{cases} 1 - e^{-\mu(\frac{t}{s} - \alpha)}, & \text{if } t \geq s\alpha, \\ 0, & \text{otherwise.} \end{cases} \tag{25}$$

The statistical model considered above is a shifted exponential distribution, such that the duration of a computation cannot be less than α. We also note that although the overall computation time at a particular worker has an exponential distribution, the duration of each computation is assumed to be identical. Let $P_s(t)$ denote the probability of completing exactly s computations by time t. We have

$$F_s(t) = \sum_{s'=s}^{r} P_{s'}(t), \tag{26}$$

where $P_r(t) = F_r(t)$, since there is a total of r computations assigned to each worker. One can observe from (26) that $P_s(t) = F_s(t) - F_{s+1}(t)$, and it can be written as follows:

$$P_s(t) = \begin{cases} 0, & \text{if } t < s\alpha, \\ 1 - e^{-\mu(\frac{t}{s} - \alpha)}, & s\alpha \leq t < (s+1)\alpha, \\ e^{-\mu(\frac{t}{s+1} - \alpha)} - e^{-\mu(\frac{t}{s} - \alpha)}, & (s+1)\alpha < t. \end{cases} \tag{27}$$

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, $s = 0, \ldots, r$, and define $\mathbf{N}(t) \triangleq (N_0(t), \ldots, N_r(t))$, where $\sum_{s=0}^{r} N_s(t) = K$. The probability of a particular realization is given by

$$\Pr(\mathbf{N}(t)) = \prod_{s=0}^{r} P_s(t)^{N_s} \binom{K - \sum_{j<s} N_j}{N_s}. \tag{28}$$

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 PS by time t, $\Pr(T \leq t)$, is given by $\Pr(M(t) \geq M_{th})$. Consequently, we have

$$\Pr(T \leq t) = \sum_{\mathbf{N}(t): M(t) \geq M_{th}} \Pr(\mathbf{N}(t)), \tag{29}$$

and

$$E[T] = \int_0^\infty \Pr(T > t) dt \tag{30}$$

$$= \int_0^\infty \left[1 - \sum_{\mathbf{N}(t): M(t) \geq M_{th}} \Pr(\mathbf{N}(t)) \right] dt. \tag{31}$$

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 server computation load r, we can have

$$\Pr(T \leq t) = \sum_{k=K_{th}}^{K} \binom{K}{k} (1 - e^{-\mu(\frac{t}{r} - \alpha)})^k (e^{-\mu(\frac{t}{r} - \alpha)})^{K-k}, \tag{32}$$

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

7. Numerical Results and Discussions

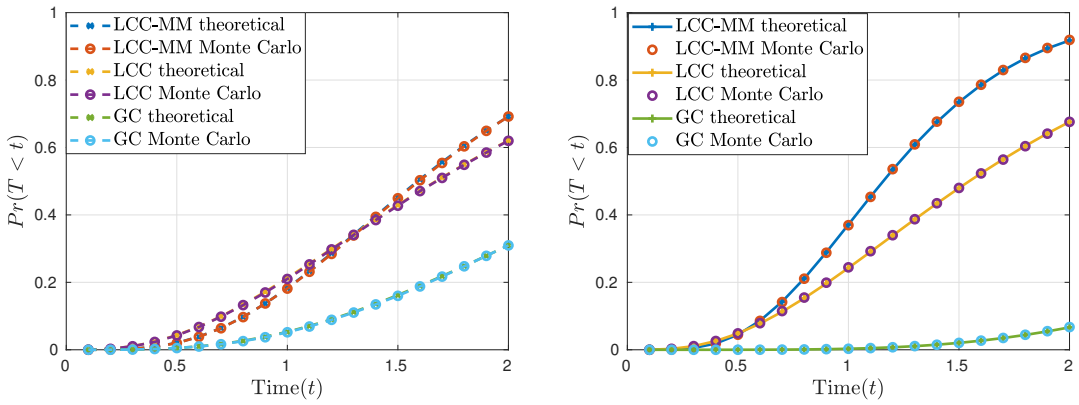
For the numerical results, we consider three different simulation setups, namely model-based, data driven and real time implementation. In the first setup, we use the shifted exponential distribution model for the computation time statistics to analyze the average completion time. For the second setup, we initialize 21 Amazon EC2 instances (where the first instance is considered to be the PS), then for each EC2 instance we measure the computation time of a certain job as well as communication time with the parameter server, over different times of the day, to form a dataset to analyze the average completion time statistics. Finally, in the third set of simulations, we conduct a real time experiment via implementing a linear regression problem on Amazon EC2 instances through 1000 iterations to monitor the average completion time statistics.

7.1. Model-Based Analysis

We first verify the correctness of the expressions provided for the per-iteration completion time statistics in (29) and (32) through Monte Carlo simulations generating 100,000 independent realizations.

Then, we will show that the MMC approach can reduce the average per-iteration completion time, $E[T]$, significantly. In particular, we analyze the per-iteration completion time of three different DGD schemes, GC, LCC, and LCC with MMC (LCC-MM). For the simulations we consider two different settings, $K = 6$, $r = 3$ and $K = 10$, $r = 5$, respectively, and use the cumulative density function (CDF) in (25) with parameters $\mu = 10$ and $\alpha = 0.01$ for the completion time statistics.

In Figure 2 we plot the CDF of the per-iteration completion time T for GC, LCC, and LCC-MM schemes according to the closed-form expressions derived in Section 6 and Monte Carlo simulations. We observe from Figure 2 that the two match perfectly. We also observe that although the LCC-MM and LCC schemes perform closely in the first scenario (Figure 2a), LCC-MM outperforms the LCC scheme in the second scenario (Figure 2b). This is because, as the computation load r increases, it takes more time for even the fast workers to complete all the assigned computations, which results in a higher number of non-persistent stragglers. Hence, the performance gap between LCC-MM and LCC increases with r. Similarly, as expected, since the non-straggling threshold of GC does not scale with K, we observe that GC performs better for small r when the K/r ratio is preserved.

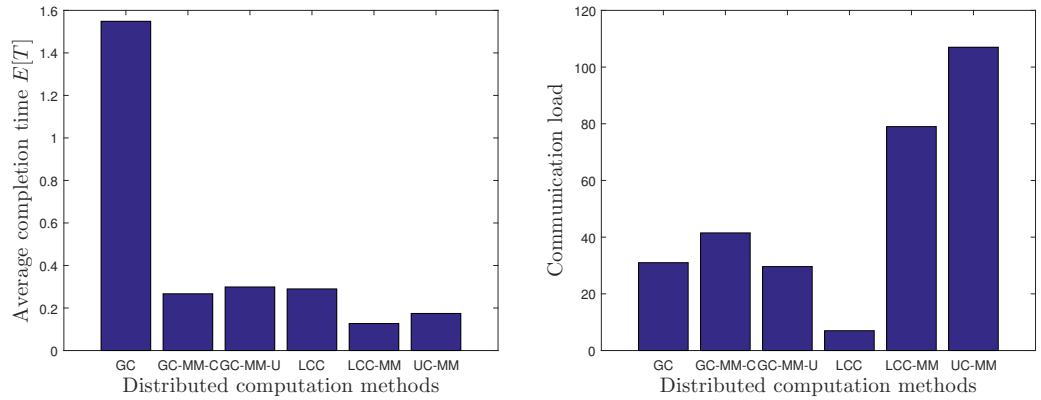


(**a**) (**b**)

Figure 2. Per-iteration completion time statistics; (**a**) $K = 6$, $r = 3$, and (**b**) $N = K$, $r = 5$.

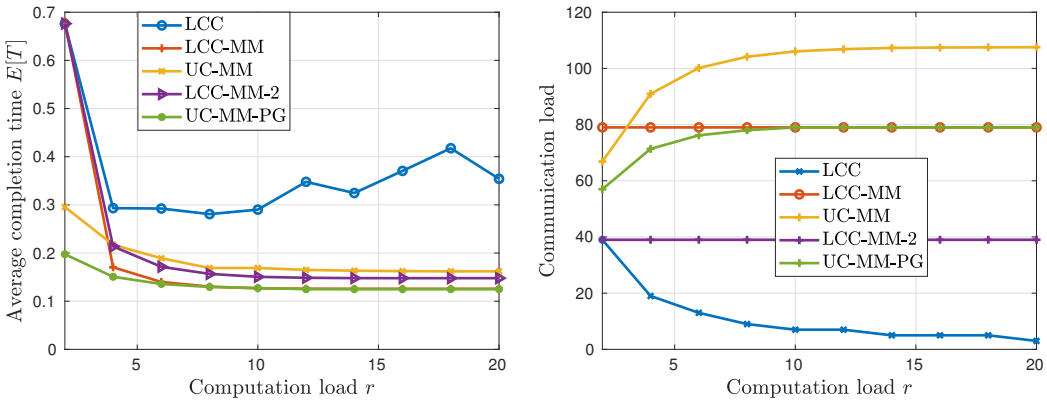
Next, we consider the setup from [20], where $K = 40$ workers are employed for DGD with a computation load of $r = 10$, and analyze the performance of six different DGD schemes, namely GC, GC with MMC and uncorrelated design (GC-MM-U), GC with MMC and correlated design (GC-MM-C), LCC, LCC-MM and uncoded computation with MMC (UC-MM). For the design of GC-MM-U, we divide the workers into four equal-size clusters, and we use uncorrelated code structure with order vector $\mathbf{m} = [6, 8, 10]$, so that each worker can send up to 3 coded partial gradients. Similarly, for the design of GC-MM-C, we again divide the workers into four equal-size clusters and use the correlated code structure with order $m = 6$, so that each worker can send up to 5 coded partial gradients. For the computation time statistics, we use the distribution in (25) with parameters $\mu = 10$ and $\alpha = 0.01$. In 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 PS, and the results obtained from 100,000 Monte Carlo realizations are illustrated in Figure 3. We observe that LCC-MM approach can provide approximately 50% reduction in the average completion time compared to LCC, and more than 90% reduction compared to GC. A more interesting result is that the UC-MM scheme outperforms both LCC and GC. This result is especially important since UC-MM has no decoding complexity at the PS. Hence, when the decoding time of PS is also included in the average per-iteration completion time this improvement will be even more significant. We also observe that LCC-MM scheme achieves the minimum average completion time. However, Figure 3b highlights that the MMC schemes, particularly LCC-MM and UC-MM, induce much higher communication load compared to the conventional single-message schemes. The results illustrated in Figure 3 also show that the multi-message variations of GC can perform as well as LCC in terms

of the average per-iteration completion time, while inducing much lower communication overhead compared to the LCC-MM and UC-MM schemes.



(**a**) (**b**)

Figure 3. Per-iteration completion time and communication load statistics; (**a**) average completion time performance, and (**b**) communication load performance.



(**a**) (**b**)

Figure 4. Per-iteration completion time and communication load statistics; (**a**) average completion vs. computation load, and (**b**) communication load vs. computation load.

Finally, on the same setup, we analyze the performance of the DGD schemes with respect to the computation load r, and compute both the average per-iteration completion time $E[T]$ and the communication load for ten different r values, i.e., $r = 2, 4, \ldots, 20$. In Section 1, we identified two main drawbacks of the single-message coded computation schemes; namely over-computation and under-utilization. In Figure 4a, these drawbacks are explicitly demonstrated. One can observe that after a certain point, the average completion time of LCC starts to increase with r, which reflects over-computation. The gap between the LCC and LCC-MM highlights under-utilization of the computation resources.

From Figure 4a, we observe that the UC-MM scheme consistently outperforms LCC for all the computation load values. More interestingly, UC-MM performs very close to LCC-MM, and for a small r, such as $r = 2$, it can even outperform LCC-MM. Hence, in terms of the computation time UC-MM can be considered to be a better option compared to LCC especially when r is low.

On the other hand, in Figure 4b we observe that in terms of the communication load the best scheme is LCC, while the UC-MM introduces the highest communication load. We also observe that the communication load of LCC-MM remains constant with r, whereas that of the LCC (UC-MM) scheme monotonically decreases (increases) with r. Accordingly, the communication load of the LCC and UC-MM schemes are closest at $r = 2$. Hence, from both Figure 4a and Figure 4b we can conclude that

when r is low, UC-MM might be a better option compared to LCC taking into account the computation time, the communication load and decoding complexity together. We also want to underline the fact that although LCC-MM achieves a lower average completion time, MMC increases the communication load as well as the decoding complexity.

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 PS. Among these three schemes, UC-MM has the lowest decoding complexity, while LCC-MM has the highest. However, as discussed in Section 4, 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-MM scheme, where two polynomials are used, denoted by LCC-MM-2 (We use notation LCC-MM-2 and LCC-2 interchangeably.). In this scheme, for given r, coded inputs correspond to evaluation of two polynomials, each of degree $N-2$, at $r/2$ different points. Each worker sends a partial result in the PS after execution of two computations, which correspond to the evaluation of these two polynomials at the same point. Since two polynomials are used, the number of transmissions is reduced by approximately half compared to LCC-MM as illustrated in Figure 4b. 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 Figure 4a.*

Another important advantage of the UC-MM scheme is its applicability to partial gradient scenario. The objective of all the straggler avoidance schemes explained in this paper is to recover the full gradient at the PS. Accordingly, with UC-MM, the PS waits until it receives all K partial gradients to terminate the iteration. However, to reduce the computation time PS may terminate an iteration after receiving only $\tilde{K} < K$ partial gradients out of K [49]. We refer to this variation of UC-MM scheme as UC-MM-PG. For the UC-MM-PG scheme, the key design parameter is the tolerance rate $\frac{K-\tilde{K}}{K}$ and for our analysis we set the tolerance rate to 5%. The results in Figure 4a show that when r is small, UC-MMC-PG can reduce the average completion time up to 70% compared to LCC, and up to 33% compared to UC-MMC; while only 2 out of 40 gradient values are missing at each iteration. In addition to an improvement in the average completion time, the UC-MMC-PG scheme can also reduce the communication load as shown in Figure 4b. We remark that in partial gradient approach the estimated gradient, due to missing partial gradients, is not the original gradient but an estimate of it. Although each update is less accurate compared to full-gradient updates, since the parameter vector is updated over many iterations, partial gradient approach may converge to the optimal value faster than the full-gradient approach. Indeed, stochastic gradient descent is an extreme case of this partial gradient approach, and is commonly used in practice. Moreover, tolerance rate can be dynamically updated through iterations to achieve better convergence results.

7.2. Data Driven Simulations

In this setup, we initialized 21 Amazon EC2 t2.micro instances, where the first one is labeled as the parameter server. We use the MPI protocol, particularly mpi4py [56], to establish connections between instances. For the computation, we consider a matrix-vector multiplication with sizes 3000×3000 and 3000×1, respectively, which is the core computation task for GD in a linear regression problem assuming that the whole dataset is divided into 20 subsets each containing 3000 data points and each data point is a vector of 3000 parameters. We measure the computation time using time.time() command before and after each computation. For message passing we use non-blocking communication with Issend and Irecv commands for message sending and receiving, respectively. Furthermore, we use wait() command to verify the time instant when the message is successfully received and again we use the time.time() command to measure the time.

For data collection, we do point-to-point analysis such that in each simulation we use only one instance and the parameter server. The chosen instance performs the computation (the assigned matrix-vector multiplication) and sends the result to the PS, which is repeated after receiving a

new vector from the PS. In total, we form a measurement set of size 3000 for both computation and communication latency for each node. These measurement sets are then used for our average per-iteration time analysis. We want to note that in practice one of the predominant factors affecting the average completion time is the congestion at the PS due to the MPI protocol; however, this is very much dependent on the particular protocol used, and can be reduced or eliminated with more efficient communication protocol. For example, by employing a hierarchical framework with multiple PSs congestion issue can be resolved in large-scale implementations. Hence, we first analyze the average completion time ignoring the effects of congestion. We refer to these simulations as *data driven*, which are based on the assumptions that the communication channels from workers to PS are orthogonal.

We consider two different scenarios. In the first scenario we randomly delay the computation time of the instances for a fixed duration. In the second simulation, in addition to computational delay, we add exponentially distributed delay to the communication latency.

7.2.1. Scenario 1

We introduce the term *delay probability*, denoted by p, to refer to the probability of a machine to be delayed. This delay can be due to the computation process, as mostly argued in the literature, a possible access failure (connection lost), or the queuing delay due to congestion of computation tasks. For the simulations, we consider a fixed additional delay that comprise all aforementioned delays, which we refer as the *initial delay*. Fixed initial delay approach have been also used for simulations in [12,20].

In our simulations, we consider failure probabilities 0.2, 0.3 and 0.4, and computational loads of $r = 4$ and $r = 6$. We use the GC-MM-C scheme with message order 3 (with cluster size of 5) and 4 (with cluster size of 10) when $r = 4$ and $r = 6$, respectively. Similarly, we use the GC-MM-U scheme with message order vector $\boldsymbol{m} = [3, 4]$ (with cluster size of 5) and $\boldsymbol{m} = [4, 5, 6]$ (with cluster size of 10) when $r = 4$ and $r = 6$, respectively. We refer to each (r, p) pair as a sub-scenario and consider six of them in total. For each sub-scenario we vary the initial delay in the range of 6 to 36 milliseconds (ms), and the results are shown in Figure 5.

From the results, an immediate observation is that multi-message schemes perform better than their single-message counterparts when the computation load r is high. We note that although a higher computation load reduces the non-straggler threshold, it also increases the computation time of the non-straggler workers. Hence, when the ratio of non-straggler threshold to the number of workers is less than $1 - p$; that is, when the delay probability is over-estimated, we observe the limitation due to over-computation, and single-message schemes performs poorly as clearly illustrated in Figure 5b. On the other hand, MMC has flexibility of either collecting fewer computations from a large set of workers, e.g., when p is low, or collecting more computations from fewer workers, e.g., when p is high. This flexibility makes MMC schemes, especially LCC-MM and UC-MM, better options compared to their single-message counterparts.

Simulation results also point out that although LCC is superior to the GC scheme, proposed variations of GC, particularly GC-MM-C, can outperform LCC in certain cases. Moreover, we observe that the correlated GC design, GC-MM-C, performs better compared to the uncorrelated design, GC-MM-C, especially when r is large.

Finally, the simulation results, especially those with $r = 4$, show that as p increases, i.e., as $1 - p$ gets close to the ratio of non-straggler threshold to the number of workers, comparative performance of the LCC scheme improves and even outperforms LCC-MM and UC-MM schemes. This observation highlights the fact that when the PS is limited to receive computations from the same subset of workers, which is the case when p is large, LCC may perform better.

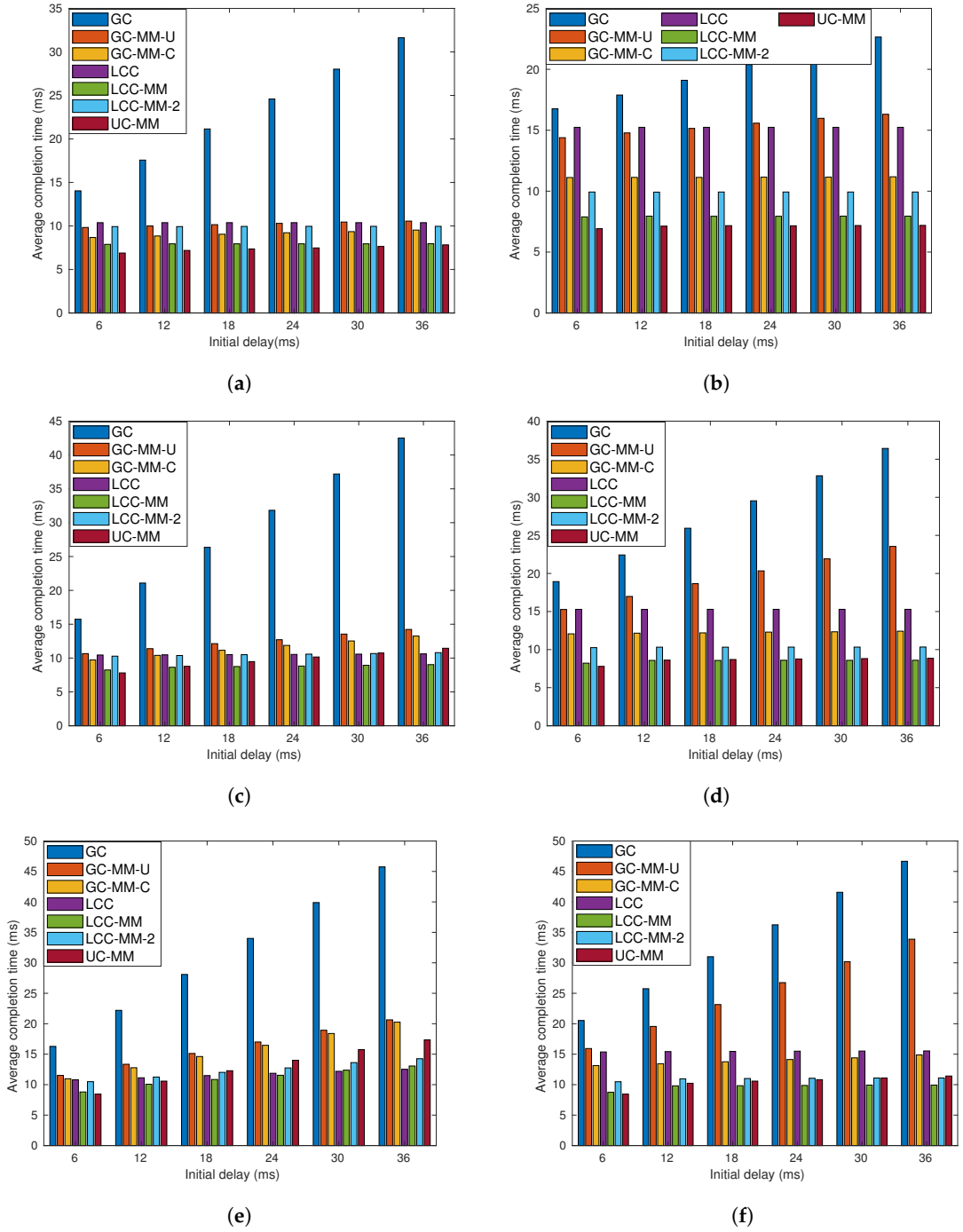


Figure 5. Average completion time analysis for GC, GC-MM-U, GC-MM-C, LCC, LCC-MM, LCC-2 and UC-MM schemes with random fixed initial delay; (**a**) $r = 4$ and $p = 0.2$, (**b**) $r = 6$ and $p = 0.2$, (**c**) $r = 4$ and $p = 0.3$, (**d**) $r = 6$ and $p = 0.3$, (**e**) $r = 4$ and $p = 0.4$, and (**f**) $r = 6$ and $p = 0.4$.

7.2.2. Scenario 2

In the previous simulations, we focus on worker-based delays by using an initial delay parameter. We remark that with non-blocking communication approach communication and computation can be executed in parallel; however, each worker can send a message when the corresponding computation is completed and the previous message is successfully received by the PS. Hence, under certain scenarios where the communication latency is higher than the computation latency MMC strategy might be inefficient. In other words, the success of the MMC strategy depends on the ratio between

the average computation and communication latency. To this end, we extend our previous analysis by adding additional exponentially distributed delays with parameter μ to the communication latency to demonstrate the impact of the communication latency on the MMC schemes.

We first set $r = 4$, and consider 4 sub-scenarios each corresponding to a different p, μ pair, where p takes values 0.2 and 0.4, and μ takes values 2 and 4. For each sub-scenario we again change the initial delay in the range of 6 to 36 ms, and the results are illustrated in Figure 6.

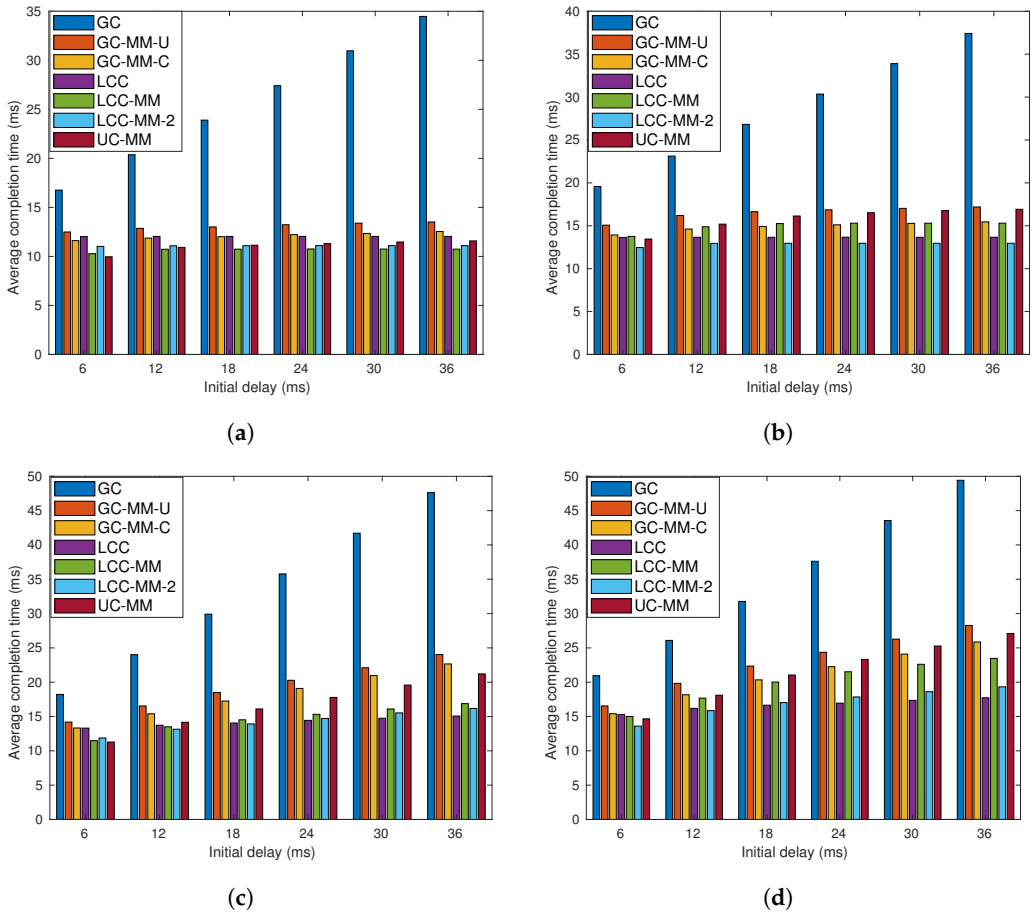


Figure 6. Average completion time analysis for GC, GC-MM-U, GC-MM-C, LCC, LCC-MM, LCC-MM-2 and UC-MM schemes with random fixed initial delay and exponential communication; (**a**) $r = 4$, $p = 0.2$ $\mu = 2$, (**b**) $r = 4$, p = 0.2 $\mu = 4$, (**c**) $r = 4$, $p = 0.4$ $\mu = 2$, and (**d**) $r = 4$, $p = 0.4$ $\mu = 4$.

One can easily observe, comparing Figure 6b and Figure 5a, that even for small p, MMC schemes, especially UC-MM, can lose their advantage over single-message schemes when the communication latency is considerably high. Indeed, LCC and its multi-message variations, LCC-MM and LCC-MM-2, outperform UC-MM in all four sub-scenarios except the first one, in which UC-MM performs slightly better than LCC. Another interesting observation is that when $\mu = 4$, GC-MM-C outperforms UC-MM especially when p is low. Hence, when the communication latency in the network is large, GC with MMC can be preferred against UC-MM.

We repeat the simulations for the same four sub-scenarios with communication load $r = 6$, and the results are illustrated in Figure 7. Although the results show similarities with the previous one, we can identify some variations. First, as we expected, the relative performance of LCC deteriorates, due to the over-computation, especially when $p = 0.2$. When $p = 0.2$, compared to the case of $r = 4$, LCC loses its advantage against UC-MM. We also observe that in none of these four sub-scenarios LCC is the best one.

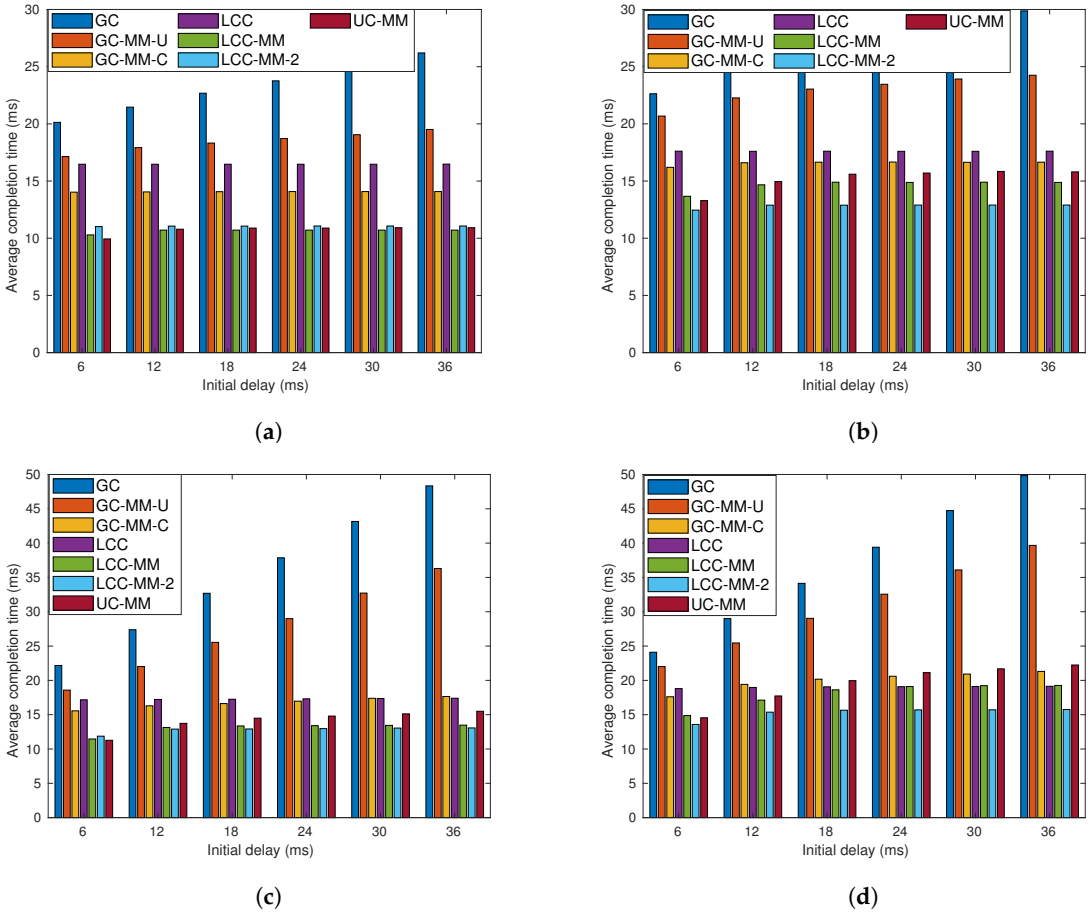


Figure 7. Average completion time analysis for GC, GC-MM-U, GC-MM-C, LCC, LCC-MM, LCC-2 and UC-MM schemes with random fixed initial delay and exponential communication delay; (**a**) $r = 6$, $p = 0.2$ and $\mu = 2$, (**b**) $r = 6$, $p = 0.2$ and $\mu = 4$, (**c**) $r = 6$, $p = 0.4$ and $\mu = 2$, and (**d**) $r = 6$, $p = 0.4$ and $\mu = 4$.

Figures 6 and 7 point out that LCC-MM-2 can be a better alternative compared to LCC, LCC-MM and UC-MMC when both the communication latency and communication load r are high. This is because LCC-MM-2, improves upon the LCC-MM and UC-MM schemes via reducing the number of messages sent at each iteration as well as increasing the time between two communication rounds which better overlaps the communication and computation processes as illustrated in Figure 8. Thanks to overlapped communication time, LCC-MM-2 scheme is more robust to communication latency compared to LCC-MM and UC-MM.

Finally, to monitor the marginal effect of the communication latency, we pick four particular cases; $p = 0.2$ with 12 ms initial delay, $p = 0.2$ with 24 ms initial delay, $p = 0.4$ with 12 ms initial delay, and $p = 0.4$ with 24 ms initial delay; and plot the performance of all the schemes with respect to μ in Figure 9. We observe that the average completion time increases with respect μ; however, while GC and LCC exhibit a gradual increase, LCC-MM and UC-MM experience a step increase with μ. In particular, when $p = 0.2$, it is clear how UC-MM and LCC-MM schemes lose their advantages with increasing communication latency.

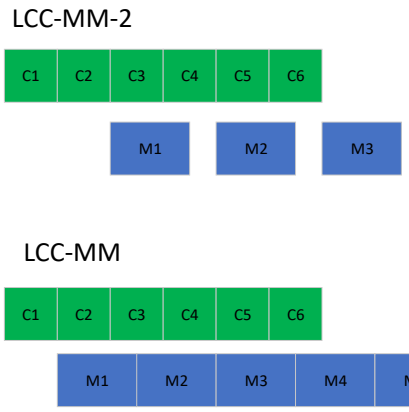


Figure 8. Overlapping communication and computation, green and blue blocks illustrate the computation and the communication steps, respectively.

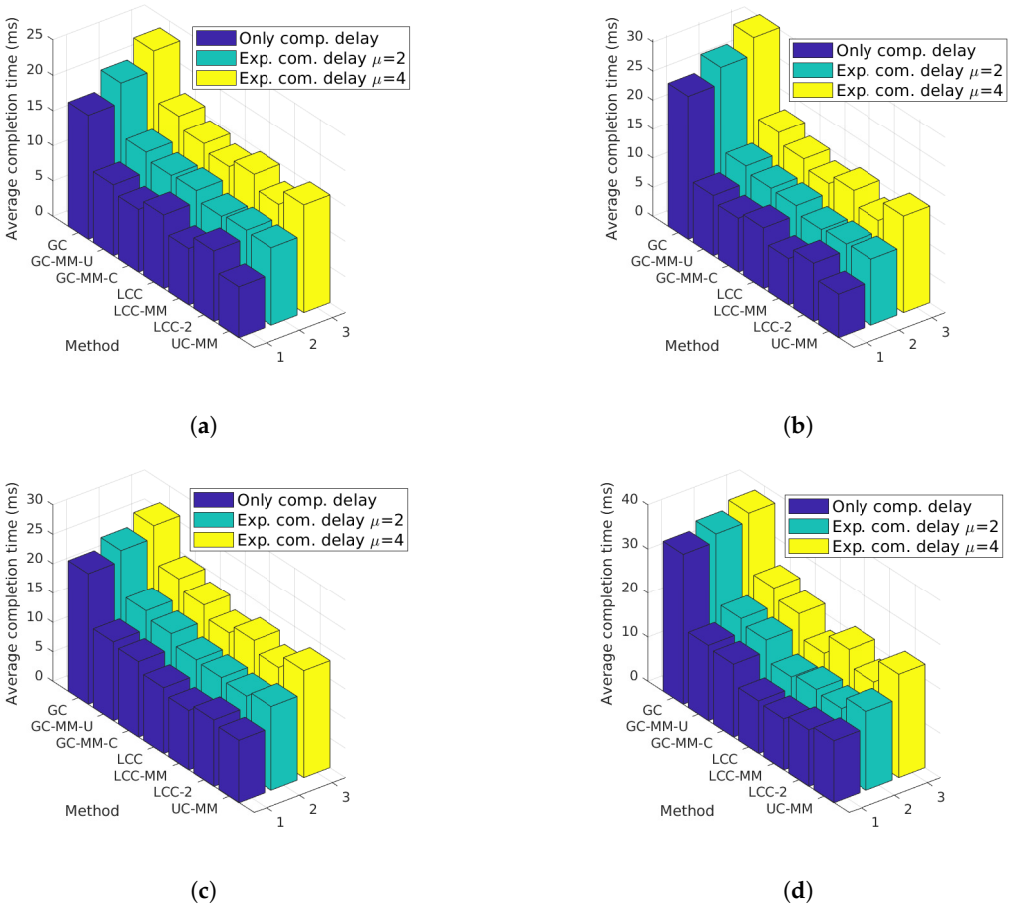


Figure 9. Per-iteration completion time for different schemes with $r = 4$; (**a**) failure probability $p = 0.2$, initial delay 12 ms, (**b**) failure probability $p = 0.2$, initial delay 24 ms, (**c**) delay probability $p = 0.4$, initial delay 12 ms, and (**d**) delay probability $p = 0.4$, initial delay 24 ms.

7.3. Real Time Simulations

Data driven simulations ignore the effect of congestion on the completion time statistics. To remedy this, we perform real time analyses on Amazon EC2 servers. As with the data driven simulations, we initialize 21 Amazon EC2 t2.micro instances, where the first one is labeled as the PS, and we use the MPI protocol to establish connections between these instances. At the beginning of each iteration, after receiving the model update from the PS, we randomly induce a fixed delay at each

instance using time.sleep() command. Then, the PS waits until the required condition to complete an iteration, which depends on the scheme employed, is met. We present the average completion time over 1000 iterations. We first set $r = 3$, and consider four different sub-scenarios with $p = 0.1, 0.2, 0.3, 0.4$. In each scenario, we change the initial delay from 6 ms to 30 ms and the results are illustrated in Figure 10.

Although GC-MM-C and UC-MM outperform LCC when both initial delay and p are low, in general, LCC achieves the best performance, especially when p is large. Nevertheless, we also observe that GC-MM-C and UC-MM perform close to LCC, particularly when p is low; indeed, in all the cases, the performance gap between LCC and UC-MM is at most 26%. Hence, when the decoding complexity of the LCC scheme as well as the initial data encoding process are taken into consideration, UC-MM scheme is still a strong candidate for distributed computation.

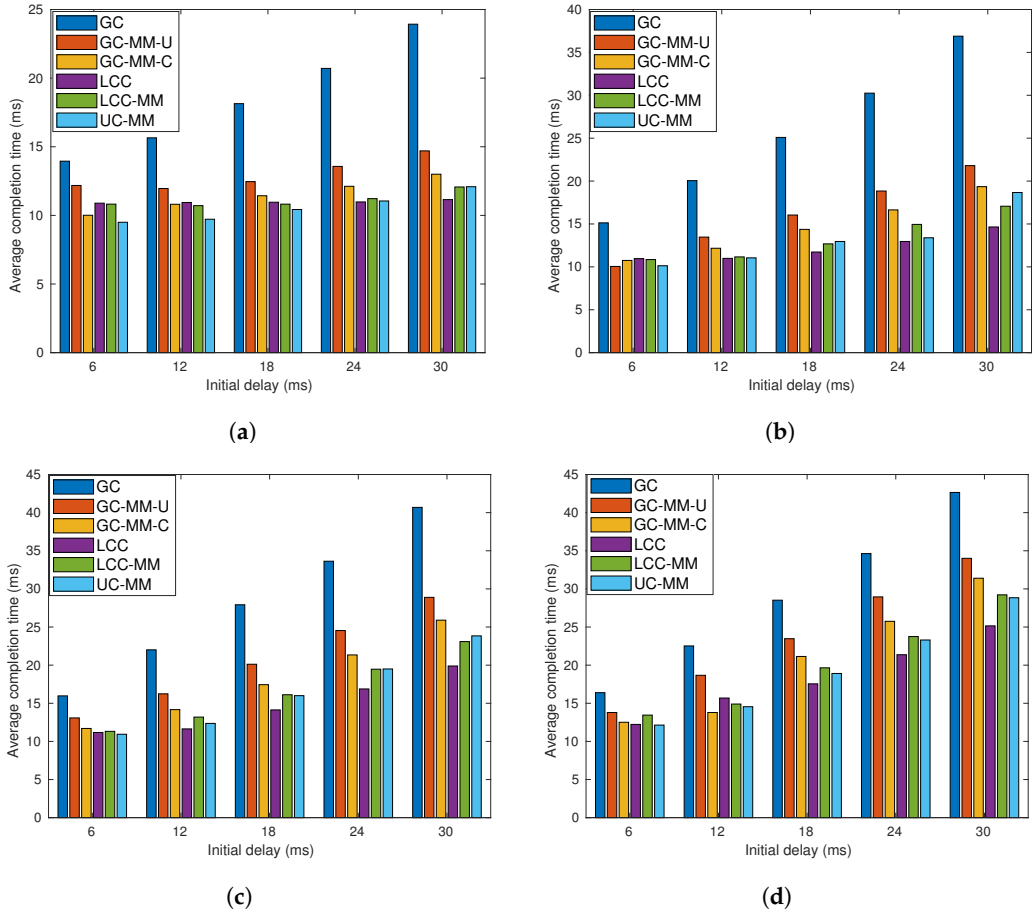


Figure 10. Per-iteration completion time for different schemes with $r = 3$; (**a**) $p = 0.1$, (**b**) $p = 0.2$, (**c**) $p = 0.3$, and (**d**) $p = 0.4$.

Next, we set $r = 4$ and repeat the simulation as in the previous case, but this time we compare the performance of the GC, LCC, LCC-MM, LCC-MM-2, and UC-MM schemes. We observe that when p is low, i.e., $p = 0.1$ and $p = 0.2$, MMC schemes LCC-MM and UC-MM outperform others. Indeed, UC-MM can perform up to 40% better compared to LCC. On the other hand, when we consider larger p values, LCC, LCC-MM, LCC-MM-2, and UC-MM schemes have similar performance, although average completion time of UC-MM scheme is slightly higher when the initial delay is large.

We remark that although the real time simulation results present similar trends with our initial data driven analysis, we observe some differences as well. In particular, when $r = 4$ and $p = 0.2$, we expect UC-MM and LCC-MM schemes to perform much better based on our data driven analysis

illustrated in Figure 5a. However, as we discussed in Section 7.2.2, communication latency is also an important factor for the average completion time statistics, and the multi-message schemes are more prone to communication delays. Our interpretation for the results in Figure 11b is that the performance of the UC-MM and LCC-MM schemes are limited due to the congestion at the PS. To show the effect of congestion more explicitly we limit our focus to two cases with initial delay of 12 ms and $p = 0.2$, and initial delay of 24 ms and $p = 0.2$. For these two cases, we compare the data driven simulation results of GC, LCC, LCC-MM, LCC-MM-2 and UC-MM schemes with their real time counterparts in Figure 12.

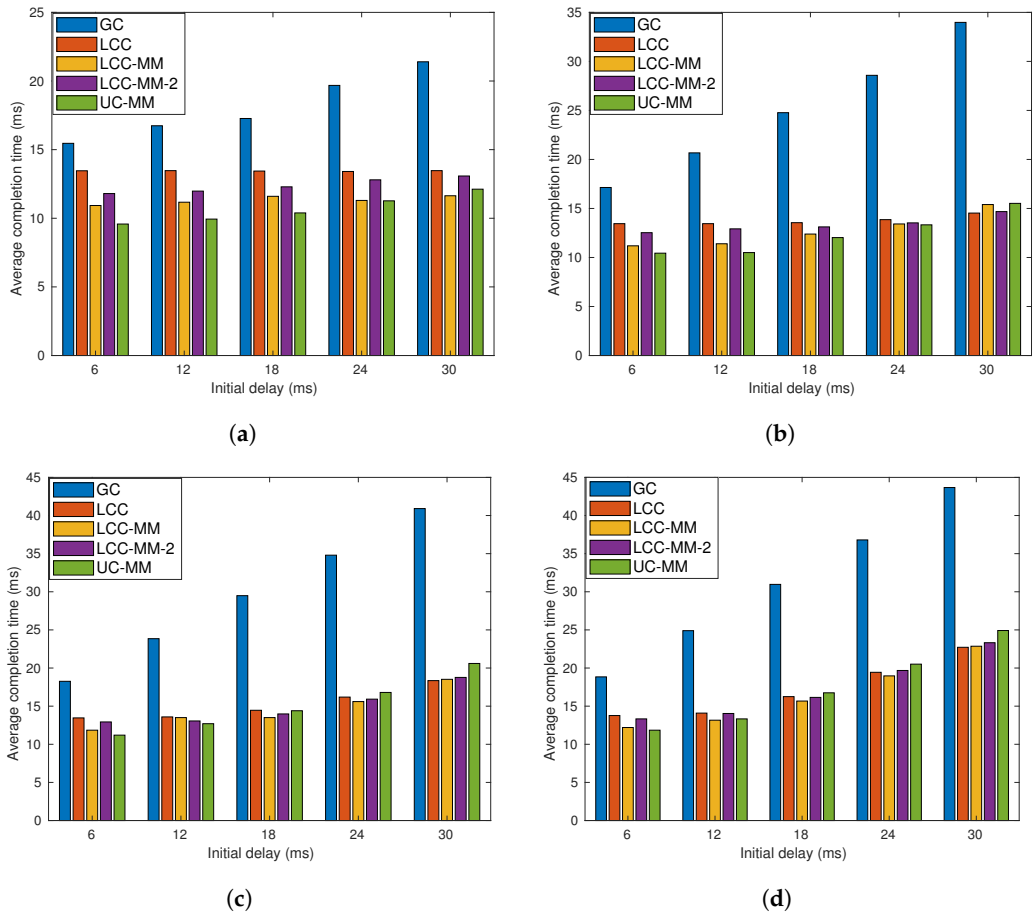


Figure 11. Per-iteration completion time for different schemes with $r = 4$; (**a**) delay probability $p = 0.1$, (**b**) delay probability $p = 0.2$, (**c**) delay probability $p = 0.3$, and (**d**) delay probability $p = 0.4$.

It is clear from Figure 12a that all the schemes suffer from the congestion, but its effect is more significant for multi-message schemes. Figure 12b, further shows that multi-message schemes, particularly LCC-MM and UC-MM, may lose their advantage due to congestion. We emphasize that these observations are consistent with our data driven simulation results with exponential communication delay.

One of the most interesting observations from the real time simulation results is the trend of the LCC schemes, particularly LCC and LCC-MM-2, with respect to initial delay. According to data driven results in Figure 5, LCC scheme should be robust to the initial delay; hence, we expect the average completion time of the LCC scheme not to increase with initial delay, but the real time simulation results in Figure 11d seem to be inconsistent with this intuition. However, this discrepancy results from the way communication delay is introduced in real time scenarios. When we introduce delay using the time.sleep() command in real time simulations, the instance might be still sleeping in the

next iteration since average completion time is less than the initial delay in general. In other words, an initial delay at a particular iteration can affect the following iterations, which is not the case in data driven simulations. This impact becomes more visible as p increases. To verify our reasoning we repeat the simulations for $p = 0.4$ with different initial delays, but this time we terminate the delay when the iteration is completed, so that the delay in one iteration has no impact on the following iterations. The corresponding simulation results are illustrated in Figure 13, which support our interpretation.

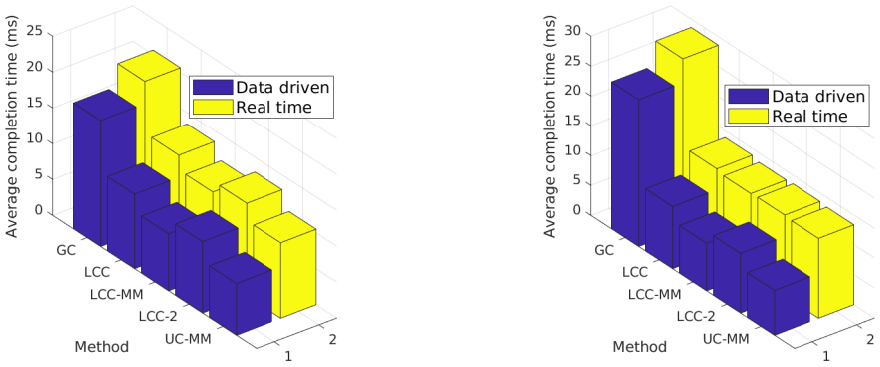


Figure 12. Comparison of data driven simulation results with real time simulation results; (**a**) $p = 0.2$ and initial delay is 12 ms, and (**b**) $p = 0.2$ and initial delay is 24 ms.

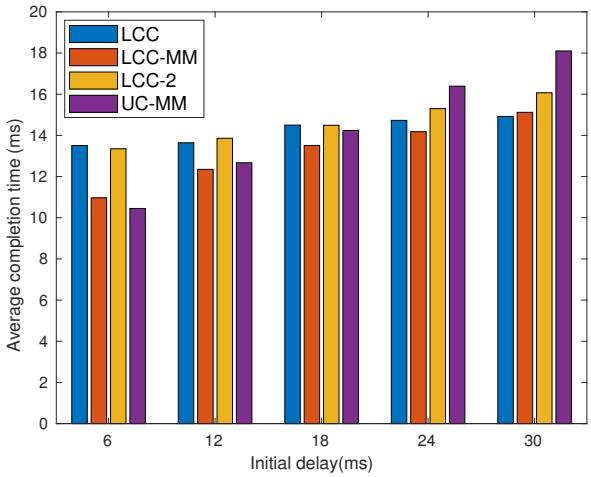


Figure 13. Performance under uncorrelated delay.

This observation leads to a new discussion on the modelling of delay at workers. In the literature, the delay is mostly associated with the computation process. In that case, one can argue that after each iteration uncompleted jobs will be terminated, so that the delay will not affect the following iterations. On the other hand, it is also possible to observe delays due to access failure or scheduled tasks for other clients. Such delays are not limited to a single iteration, causing correlation among delays over consecutive iterations.

Next, we consider both correlated and uncorrelated delays for completeness of our analysis. We set $r = 6$, and for both correlated and uncorrelated scenario we analyze two cases with $p = 0.1$ and $p = 0.3$. The corresponding results are illustrated in Figure 14. In the case of correlated delay with $p = 0.1$, compared to LCC, LCC-MM and UC-MM can achieve 36–40% and 42–58% reduction in the average completion time, respectively. Similarly, they achieve around 48% and 60% reduction, respectively, when the delay is uncorrelated. When the delay is uncorrelated, we observe similar

trends for $p = 0.3$, such that both LCC-MM and UC-MM achieve around 40% reduction in the average completion time compared to LCC. When the delay is correlated, UC-MM still outperforms LCC, but its performance deteriorates with the increase in the initial delay and when the initial delay is large LCC-MM becomes a better option. We believe that understanding the impact of correlation in the delay over time is an interesting future research direction.

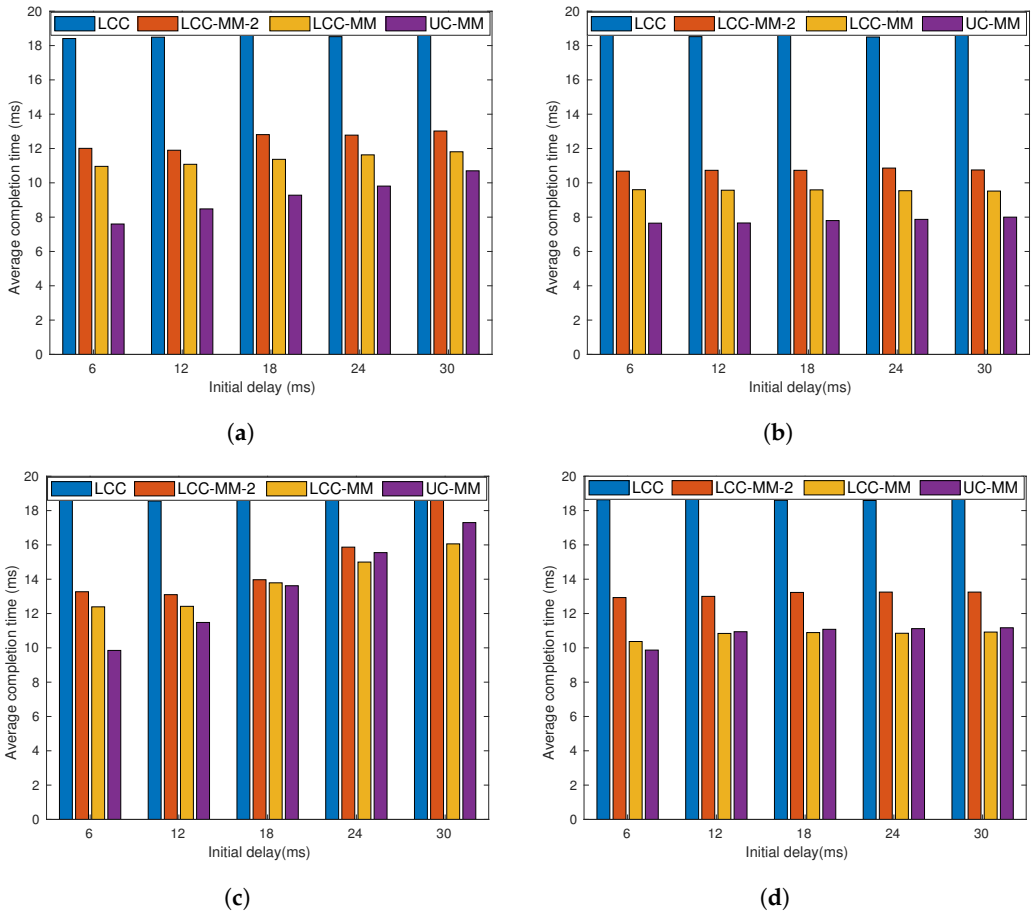


Figure 14. Per-iteration completion time for different schemes with $r = 6$; (**a**) correlated delay, $p = 0.1$, (**b**) uncorrelated delay, $p = 0.1$ (**c**) correlated delay, $p = 0.3$, and (**d**) uncorrelated delay, $p = 0.3$.

7.4. Discussions

Comparing data driven and real time simulation results, we have shown that network congestion, especially in large-scale implementations, might be a predominant issue for the performance. For real time implementations, we expect the communication latency to scale with the number of instances which limits the advantage of the MMC approach. However, a hierarchical network architecture, where the instances are grouped and multiple PSs are employed, can be used to alleviate the congestion, and thus MMC approach can still be beneficial.

In this paper, we have mostly limited our focus to the exact recovery of the full gradient; however partial gradient recovery, as well as gradient approximation, are both important research directions, which have been recently studied by several works [50–54]. We remark that partial gradient or approximate gradient recovery, can reduce the computation time by allowing less accurate updates. Moreover, the most popular optimization framework for *deep learning* is SGD, which basically uses *unbiased estimation* of the gradient using randomly sampled data [57]. Therefore, one can argue that the full gradient may not be required for a successful implementation of the GD framework in many machine learning applications. However, as already discussed in [12], missing partial

gradients may cause GD algorithm to diverge in some cases, particularly when an acceleration strategy, such as Nesterov's accelerated gradient, is employed. In addition, even for the SGD implementation, it is shown that the number of required iterations for training can be reduced by increasing the batch size [58], which is actually the main motivation behind large-scale implementations [59,60]. Furthermore, impact of the stragglers on the convergence may also depend on the dataset, its distribution among the workers (such as i.i.d./non-i.i.d. distributions) and the straggler realizations.

Finally, we want to note that in this paper, for the overall latency analysis we take into account the computation time and the communication time, but ignore the latency at PS due to the encoding complexity. As discussed in [14], the implemented code structure also plays an important role in the overall latency. However, in the scope of this paper, our focus has been to introduce a design framework for distributed learning with MMC, and we also note that different code structures can be incorporated with the introduced framework. Hence, we leave the MMC strategy with reduced decoding complexity as a future extension of this work.

8. Conclusions

We have introduced novel coded and uncoded DGD schemes when MMC is allowed from each worker at each iteration. First, we have 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 have 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 exploiting non-straggling workers at the expense of an increase in the average communication load. We have also observed that UC-MM 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. In this paper, we have performed comprehensive simulations with different parameters to highlight the fundamental trade-offs in the practical implementation of the distributed computation in the context of gradient descent for machine learning applications.

Author Contributions: Conceptualization, E.O., D.G. and S.U.; methodology, E.O., D.G. and S.U.; writing–original draft preparation, E.O.; writing–review and editing, D.G. and S.U.; funding acquisition, D.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Marie Sklodowska-Curie Action SCAVENGE (grant agreement no. 675891), and by the European Research Council (ERC) Starting Grant BEACON (grant agreement no. 677854).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dean, J.; Corrado, G.S.; Monga, R.; Chen, K.; Devin, M.; Le, Q.V.; Mao, M.Z.; Ranzato, M.; Senior, A.; Tucker, P.; et al. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*; Curran Associates Inc.: New York, NY, USA, 2012; Volume 1, pp. 1223–1231.
2. Dekel, O.; Gilad-Bachrach, R.; Shamir, O.; Xiao, L. Optimal Distributed Online Prediction Using Mini-batches. *J. Mach. Learn. Res.* **2012**, *13*, 165–202.
3. Zinkevich, M.A.; Weimer, M.; Smola, A.; Li, L. Parallelized Stochastic Gradient Descent. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems*; Curran Associates Inc.: Red Hook, NY, USA, 2010; Volume 2, pp. 2595–2603.

4. Li, M.; Andersen, D.G.; Park, J.W.; Smola, A.J.; Ahmed, A.; Josifovski, V.; Long, J.; Shekita, E.J.; Su, B.Y. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*; USENIX Association: Berkeley, CA, USA, 2014; pp. 583–598.
5. Li, S.; Kalan, S.M.M.; Avestimehr, A.S.; Soltanolkotabi, M. Near-Optimal Straggler Mitigation for Distributed Gradient Methods. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops, Vancouver, BC, Canada, 21–25 May 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 857–866.
6. Ferdinand, N.; Draper, S.C. Anytime Stochastic Gradient Descent: A Time to Hear from all the Workers. In Proceedings of the 2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton), Monticello, IL, USA, 2–5 October 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 552–559, doi:10.1109/ALLERTON.2018.8635903. [CrossRef]
7. Mohammadi Amiri, M.; Gündüz, D. Computation Scheduling for Distributed Machine Learning with Straggling Workers. *IEEE Trans. Signal Process.* **2019**, *67*, 6270–6284, doi:10.1109/TSP.2019.2952051. [CrossRef]
8. Behrouzi-Far, A.; Soljanin, E. On the Effect of Task-to-Worker Assignment in Distributed Computing Systems with Stragglers. In Proceedings of the 2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton), Monticello, IL, USA, 2–5 October 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 560–566, doi:10.1109/ALLERTON.2018.8636064. [CrossRef]
9. Chen, J.; Monga, R.; Bengio, S.; Józefowicz, R. Revisiting Distributed Synchronous SGD. *arXiv* **2016**, arXiv:1604.00981.
10. Aktas, M.F.; Soljanin, E. Straggler Mitigation at Scale. *arXiv* **2019**, arXiv:1906.10664, doi:10.1109/TNET.2019.2946464.
11. Wang, D.; Joshi, G.; Wornell, G.W. Efficient Straggler Replication in Large-Scale Parallel Computing. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2019**, *4*, 2376–3639, doi:10.1145/3310336. [CrossRef]
12. Tandon, R.; Lei, Q.; Dimakis, A.G.; Karampatziakis, N. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *Proceedings of the 34th International Conference on Machine Learning*; Precup, D., Teh, Y.W., Eds.; PMLR International Convention Centre: Sydney, Australia, 2017; Volume 70, pp. 3368–3376.
13. Ye, M.; Abbe, E. Communication-Computation Efficient Gradient Coding. In *Proceedings of the 35th International Conference on Machine Learning*; Dy, J., Krause, A., Eds.; PMLR: Stockholmsmässan, Stockholm, Sweden, 2018; Volume 80, pp. 5610–5619.
14. Halbawi, W.; Azizan, N.; Salehi, F.; Hassibi, B. Improving Distributed Gradient Descent Using Reed-Solomon Codes. In Proceedings of the 2018 IEEE International Symposium on Information Theory (ISIT), Vail, CO, USA, 17–22 June 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 2027–2031, doi:10.1109/ISIT.2018.8437467. [CrossRef]
15. Ozfatura, E.; Gündüz, D.; Ulukus, S. Gradient Coding with Clustering and Multi-Message Communication. In *2019 IEEE Data Science Workshop (DSW)*; IEEE: Piscataway, NJ, USA, 2019; pp. 42–46, doi:10.1109/DSW.2019.8755563. [CrossRef]
16. Sasi, S.; Lalitha, V.; Aggarwal, V.; Rajan, B.S. Straggler Mitigation with Tiered Gradient Codes. *arXiv* **2019**, arXiv:1909.02516.
17. Lee, K.; Lam, M.; Pedarsani, R.; Papailiopoulos, D.; Ramchandran, K. Speeding Up Distributed Machine Learning Using Codes. *IEEE Trans. Inf. Theory* **2018**, *64*, 1514–1529. [CrossRef]
18. Ferdinand, N.; Draper, S.C. Hierarchical Coded Computation. In Proceedings of the IEEE International Symposium on Information Theory (ISIT), Vail, CO, USA, 17–22 June 2018; pp. 1620–1624.
19. Maity, R.K.; Singh Rawa, A.; Mazumdar, A. Robust Gradient Descent via Moment Encoding and LDPC Codes. In Proceedings of the IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 2734–2738. doi:10.1109/ISIT.2019.8849514. [CrossRef]
20. Li, S.; Kalan, S.M.M.; Yu, Q.; Soltanolkotabi, M.; Avestimehr, A.S. Polynomially Coded Regression: Optimal Straggler Mitigation via Data Encoding. *arXiv* **2018**, arXiv:1805.09934.
21. Ozfatura, E.; Gündüz, D.; Ulukus, S. Speeding Up Distributed Gradient Descent by Utilizing Non-persistent Stragglers. In Proceedings of the IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 2729–2733, doi:10.1109/ISIT.2019.8849684. [CrossRef]
22. Dutta, S.; Fahim, M.; Haddadpour, F.; Jeong, H.; Cadambe, V.; Grover, P. On the Optimal Recovery Threshold of Coded Matrix Multiplication. *IEEE Trans. Inf. Theory* **2019**, *66*, 278–301. [CrossRef]

23. Yu, Q.; Maddah-Ali, M.; Avestimehr, S. Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication. In *Advances in Neural Information Processing Systems 30*; Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2017; pp. 4403–4413.
24. Park, H.; Lee, K.; Sohn, J.; Suh, C.; Moon, J. Hierarchical Coding for Distributed Computing. In Proceedings of the 2018 IEEE International Symposium on Information Theory (ISIT), Vail, CO, USA, 17–22 June 2018; pp. 1630–1634.
25. Mallick, A.; Chaudhari, M.; Joshi, G. Fast and Efficient Distributed Matrix-vector Multiplication Using Rateless Fountain Codes. In Proceedings of the ICASSP 2019—2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brighton, UK, 12–17 May 2019; pp. 8192–8196.
26. Karakus, C.; Sun, Y.; Diggavi, S.; Yin, W. Straggler Mitigation in Distributed Optimization Through Data Encoding. In *Advances in Neural Information Processing Systems 30*; Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2017; pp. 5434–5442.
27. Kiani, S.; Ferdinand, N.; Draper, S.C. Exploitation of Stragglers in Coded Computation. In Proceedings of the 2018 IEEE International Symposium on Information Theory (ISIT), Vail, CO, USA, 17–22 June 2018; pp. 1988–1992.
28. Das, A.B.; Tang, L.; Ramamoorthy, A. C3LES: Codes for Coded Computation that Leverage Stragglers. In Proceedings of the 2018 IEEE Information Theory Workshop (ITW), Guangzhou, China, 25–29 November 2018; pp. 1–5.
29. Ozfatura, E.; Ulukus, S.; Gündüz, D. Distributed Gradient Descent with Coded Partial Gradient Computations. In Proceedings of the ICASSP 2019—2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brighton, UK, 12–17 May 2019; pp. 3492–3496.
30. Haddadpour, F.; Yang, Y.; Chaudhari, M.; Cadambe, V.R.; Grover, P. Straggler-Resilient and Communication-Efficient Distributed Iterative Linear Solver. *arXiv* **2018**, arXiv:1806.06140.
31. Wang, H.; Guo, S.; Tang, B.; Li, R.; Li, C. Heterogeneity-aware Gradient Coding for Straggler Tolerance. In Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Dallas, TX, USA, 7–10 July 2019; pp. 555–564.
32. Kim, M.; Sohn, J.; Moon, J. Coded Matrix Multiplication on a Group-Based Model. In Proceedings of the 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 722–726, doi:10.1109/ISIT.2019.8849317. [CrossRef]
33. Yang, Y.; Interlandi, M.; Grover, P.; Kar, S.; Amizadeh, S.; Weimer, M. Coded Elastic Computing. In Proceedings of the 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 2654–2658.
34. Yu, Q.; Maddah-Ali, M.A.; Avestimehr, A.S. Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding. In Proceedings of the 2018 IEEE International Symposium on Information Theory (ISIT), Vail, CO, USA, 17–22 June 2018; pp. 2022–2026.
35. Dutta, S.; Bai, Z.; Jeong, H.; Low, T.M.; Grover, P. A Unified Coded Deep Neural Network Training Strategy based on Generalized PolyDot codes. In Proceedings of the 2018 IEEE International Symposium on Information Theory (ISIT), Vail, CO, USA, 17–22 June 2018; pp. 1585–1589.
36. Soto, P.; Li, J.; Fan, X. Dual Entangled Polynomial Code: Three-Dimensional Coding for Distributed Matrix Multiplication. In *Proceedings of the 36th International Conference on Machine Learning*; Chaudhuri, K., Salakhutdinov, R., Eds.; PMLR: Long Beach, CA, USA, 2019; Volume 97, pp. 5937–5945.
37. Park, H.; Moon, J. Irregular Product Coded Computation for High-Dimensional Matrix Multiplication. In Proceedings of the 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 1782–1786.
38. Das, A.B.; Ramamoorthy, A. Distributed Matrix-Vector Multiplication: A Convolutional Coding Approach. In Proceedings of the 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 3022–3026.
39. Mallick, A.; Joshi, G. Rateless Codes for Distributed Computations with Sparse Compressed Matrices. In Proceedings of the 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 2793–2797.

40. Yu, Q.; Maddah-Ali, M.A.; Avestimehr, A.S. Coded Fourier Transform. In Proceedings of the 2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton), Monticello, IL, USA, 3–6 October 2017.
41. Reisizadeh, A.; Prakash, S.; Pedarsani, R.; Avestimehr, A.S. CodedReduce: A Fast and Robust Framework for Gradient Aggregation in Distributed Learning. *arXiv* **2019**, arXiv:1902.01981.
42. Buyukates, B.; Ulukus, S. Timely Distributed Computation with Stragglers. *arXiv* **2019**, arXiv:1910.03564.
43. Hasircioglu, B.; Gomez-Vilardebo, J.; Gunduz, D. Bivariate Polynomial Coding for Exploiting Stragglers in Heterogeneous Coded Computing Systems. *arXiv* **2020**, arXiv:2001.07227.
44. Severinson, A.; i Amat, A.G.; Rosnes, E.; Lázaro, F.; Liva, G. A Droplet Approach Based on Raptor Codes for Distributed Computing with Straggling Servers. In Proceedings of the 2018 IEEE 10th International Symposium on Turbo Codes Iterative Information Processing (ISTC), Hong Kong, China, 3–7 December 2018; pp. 1–5.
45. Severinson, A.; Graell i Amat, A.; Rosnes, E. Block-Diagonal and LT Codes for Distributed Computing with Straggling Servers. *IEEE Trans. Commun.* **2019**, *67*, 1739–1753. [CrossRef]
46. Zhang, J.; Simeone, O. Improved Latency-communication Trade-off for Map-shuffle-reduce Systems with Stragglers. In Proceedings of the ICASSP 2019—2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brighton, UK, 12–17 May 2019; pp. 8172–8176.
47. Li, S.; Maddah-Ali, M.A.; Avestimehr, A.S. Coded Distributed Computing: Straggling Servers and Multistage Dataflows. In Proceedings of the 2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton), Monticello, IL, USA, 27–30 September 2016; pp. 164–171.
48. Konstantinidis, K.; Ramamoorthy, A. CAMR: Coded Aggregated MapReduce. In Proceedings of the 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 1427–1431.
49. Dutta, S.; Joshi, G.; Ghosh, S.; Dube, P.; Nagpurkar, P. Slow and Stale Gradients Can Win the Race: Error-Runtime Trade-offs in Distributed SGD. In Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS), Playa Blanca, Spain, 9–11 April 2018.
50. Bitar, R.; Wootters, M.; Rouayheb, S.E. Stochastic Gradient Coding for Straggler Mitigation in Distributed Learning. *arXiv* **2019**, arXiv:1905.05383.
51. Wang, H.; Charles, Z.B.; Papailiopoulos, D.S. ErasureHead: Distributed Gradient Descent without Delays Using Approximate Gradient Coding. *arXiv* **2019**, arXiv:1901.09671.
52. Wang, S.; Liu, J.; Shroff, N.B. Fundamental Limits of Approximate Gradient Coding. *arXiv* **2019**, arXiv:1901.08166.
53. Horii, S.; Yoshida, T.; Kobayashi, M.; Matsushima, T. Distributed Stochastic Gradient Descent Using LDGM Codes. In Proceedings of the 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 7–12 July 2019; pp. 1417–1421.
54. Zhang, J.; Simeone, O. LAGC: Lazily Aggregated Gradient Coding for Straggler-Tolerant and Communication-Efficient Distributed Learning. *arXiv* **2019**, arXiv:1905.09148.
55. Chen, T.; Giannakis, G.B.; Sun, T.; Yin, W. LAG: Lazily Aggregated Gradient for Communication-efficient Distributed Learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*; Curran Associates Inc.: Red Hook, NY, USA, 2018; pp. 5055–5065.
56. New York University. Python MPI. 2017. Available online: https://nyu-cds.github.io/python-mpi/ (accessed on 11 May 2020).
57. Bottou, L.; Curtis, F.; Nocedal, J. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.* **2018**, *60*, 223–311. [CrossRef]
58. Shallue, C.J.; Lee, J.; Antognini, J.; Sohl-Dickstein, J.; Frostig, R.; Dahl, G.E. Measuring the Effects of Data Parallelism on Neural Network Training. *J. Mach. Learn. Res.* **2019**, *20*, 1–49.

59. Goyal, P.; Dollár, P.; Girshick, R.B.; Noordhuis, P.; Wesolowski, L.; Kyrola, A.; Tulloch, A.; Jia, Y.; He, K. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv* **2017**, arXiv:1706.02677.
60. You, Y.; Zhang, Z.; Hsieh, C.J.; Demmel, J.; Keutzer, K. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing*; ACM: New York, NY, USA, 2018; pp. 1–10. doi:10.1145/3225058.3225069. [CrossRef]