2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)
Allerton Park and Retreat Center
Monticello, IL, USA, October 2-5, 2018

Cross-Iteration Coded Computing

Farzin Haddadpour, Yaoqing Yang, Viveck Cadambe, Pulkit Grover

***Abstract*—We introduce the idea of *cross-iteration coded computing*, an approach to reducing communication costs for a large class of distributed iterative algorithms involving linear operations, including gradient descent and accelerated gradient descent for quadratic loss functions. The state-of-the-art approach for these iterative algorithms involves performing one iteration of the algorithm per round of communication among the nodes. In contrast, our approach performs multiple iterations of the underlying algorithm in a single round of communication by incorporating some redundancy storage and computation. Our algorithm works in the master-worker setting with the workers storing carefully constructed linear transformations of input matrices and using these matrices in an iterative algorithm, with the master node inverting the effect of these linear transformations. In addition to reduced communication costs, a trivial generalization of our algorithm also includes resilience to stragglers and failures. The degree of redundancy of our algorithm can be tuned based on the amount of communication and straggler resilience required. Finally, we also describe a variant of our algorithm that can flexibly recover the results based on the degree of straggling in the worker nodes. The variant allows for the performance to degrade gracefully as the number of successful (non-straggling) workers is lowered.**

I. Introduction

In this work, we design a novel coded computing technique called *cross-iteration coded computing* to reduce the communication costs of state-of-the-art iterative computing by carefully introducing redundancy to storage and computation. Specifically, we focus on reducing the number of communication rounds, which is often the bottleneck in distributed computing [1]–[5]. Compared to existing coded computing techniques, the proposed technique *jointly* codes the computation of *multiple iterations*, instead of coding the computation of each iteration separately (hence the name). It shares an intellectual flavor to the attempt in [6] to break the inherently serial nature of iterative computing and parallelizing it, but through the use of coding techniques which provide rigorous theoretical guarantees. Cross-iteration coded computing can reduce several rounds of communications to only one, thereby significantly reducing the communication costs, with controlled increases in storage and computation costs. Cross-iteration coded computing can also be readily adapted to provide straggler mitigation in distributed algorithms through trivial generalizations.

We summarize the main contributions of the paper as the following:

Y. Yang and P. Grover are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA-15213. F. Haddadpour and V. Cadambe are with the Department of Electrical Engineering, Pennsylvania State University, University Park, PA 16802. The authors can be contacted at (yyaoqing,pgrover)@andrew.cmu.edu, fxh18@psu.edu, viveck@engr.psu.edu

- We provide a coded computing technique that codes jointly across several iterations of distributed computing, thereby reducing the number of communication rounds to only one by leveraging storage and computation redundancy. We provide analytical insights which demonstrate that the provided technique reduces the communication cost compared to a standard baseline algorithm. We also describe a variant that can trade-off redundancy in storage and computation, with communication cost.
- We show that the proposed technique can also provide tolerance to stragglers. A novel aspect of our technique is that it can recover the results under a computation deadline, by flexibly choosing the number of iterations in a post-processing phase based on the number of stragglers. As a consequence, the performance degrades gracefully as the number of successful workers is lowered.

A. Motivation: Distributed Implementation of Iterative Algorithms

The focus of our paper is a generic iterative algorithm, whose inputs are a $N \times 1$ vectors $\mathbf{y}, \mathbf{x}^{(0)}$ and output at the n-th iteration is represented as the $N \times 1$ vector $\mathbf{x}^{(n)}$ for $n = 1, 2, \ldots$. The algorithms we consider are parametrized by an $N \times N$ matrix $\mathbf{A}$, a sequence of real-valued degree ℓ polynomials $p^{(\ell)}(z), q^{(\ell)}(z)$, for $\ell = 1, 2, \ldots$. The iterative algorithms we consider are expressed as

$$\mathbf{x}^{(t)} = p^{(t)}(A)\mathbf{x}^{(0)} + q^{(t)}(A)\mathbf{y}. \tag{1}$$

We denote

$$p^{(t)}(z) = \sum_{i=0}^{t} \alpha_i^{(t)} z^i \quad , \quad q^{(t)}(z) = \sum_{i=0}^{t} \beta_i^{(t)} z^i, \tag{2}$$

where $\alpha_i^{(t)}, \beta_i^{(t)} \in \mathbb{R}$ for $0 \leq i \leq t, t = 1, 2, \ldots,$. Several important iterative optimization approaches, in particular for quadratic cost functions, are specializations of the above generic algorithm. Often times, these algorithms have the following recursive structure,

$$\mathbf{x}^{(t)} = \mathbf{A}\left(\sum_{i=0}^{t-1} \gamma_{i,1}^{(t)} \mathbf{x}^{(i)} + \delta_1^{(t)} \mathbf{y}\right) + \sum_{i=0}^{t-1} \gamma_{i,2}^{(t)} \mathbf{x}^{(i)} + \delta_2^{(t)} \mathbf{y}$$

which results in the following recursive structure for the polynomials:

$$p^{(t)}(z) = z \sum_{i=0}^{t-1} \left(\gamma_{i,1}^{(t)} p^{(i)}(z)\right) + \sum_{i=0}^{t-1} \left(\gamma_{i,2}^{(t)} p^{(i)}(z)\right) \tag{3}$$

978-1-5386-6596-1/18/$31.00 ©2018 IEEE

$$q^{(t)}(z) = z\left(\sum_{i=0}^{t-1}\gamma_{i,1}^{(t)}q^{(i)}(z)+\delta_1^{(t)}\right)+\sum_{i=0}^{t-1}\gamma_{i,2}^{(t)}q^{(i)}(z)+\delta_2^{(t)}. \tag{4}$$

We provide two example of such algorithms, namely gradient descent and accelerated gradient descent.

Example 1: Gradient Descent with a quadratic loss function: For a differentiable loss function $f(\mathbf{x},\mathbf{b})$, consider the iterative gradient descent algorithm that approximates $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x},\mathbf{b})$ as follows:

$$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} - \phi\nabla_{\mathbf{x}} f(\mathbf{x}^{(t-1)},\mathbf{b}),$$

where $\phi \in \mathbb{R}$ is the step size. For a quadratic loss function $f(\mathbf{x},\mathbf{b}) = ||\mathbf{M}\mathbf{x}-\mathbf{b}||^2$, the above iterative algorithm specializes as $\mathbf{x}^{(t)} = \mathbf{A}\mathbf{x}^{(t-1)} + \mathbf{y}$, where $\mathbf{A} = \mathbf{I} - 2\phi\mathbf{M}^T\mathbf{M}$ and $\mathbf{y} = 2\phi\mathbf{M}\mathbf{b}$. Note that this algorithm can be written in the form (1),(3),(4), where $p^{(t)}(z) = zp^{(t-1)}(z), p^{(1)}(z) = 1, q^{(1)}(z) = \mathbf{y}$, and $q^{(t)}(z) = zq^{(t-1)}(z) + q^{(1)}(z)$.

Example 2: Accelerated Gradient Descent: Consider the Nesterov Accelerated Gradient Descent algorithm:

$$\mathbf{v}^{(t+1)} = \mathbf{x}^{(t)} - \phi\nabla f(\mathbf{x}^{(t)})$$
$$\mathbf{x}^{(t+1)} = \mathbf{v}^{(t+1)} + \mu(\mathbf{v}^{(t+1)} - \mathbf{v}^{(t)}).$$

The above can be simplified as

$$\mathbf{x}^{(t+1)} = (\mu+1)\mathbf{x}^{(t)} - \mu\mathbf{x}^{(t-1)} - (\mu+1)\phi\nabla f(\mathbf{x}^{(t)}) - \mu\phi\nabla f(\mathbf{x}^{(t-1)}).$$

For a quadratic cost function as before, $f(\mathbf{x},\mathbf{b}) = ||\mathbf{M}\mathbf{x}-\mathbf{b}||^2$, the above iterative algorithm specializes to

$$\mathbf{x}^{(t+1)} = (1+\mu)\mathbf{A}\mathbf{x}^{(t)} - \mu\mathbf{A}\mathbf{x}^{(t-1)} + \mathbf{y}, \tag{5}$$

where $\mathbf{A} = \mathbf{I} - 2\phi\mathbf{M}^T\mathbf{M}$ and $\mathbf{y} = 2\phi\mathbf{M}\mathbf{b}$. This is equivalent to the recursion in (1), (3), (4) with $p^{(t+1)} = (1+\mu)zp^{(t)}(z) - \mu zp^{(t-1)}(z)$ and $q^{(t+1)} = (1+\mu)zq^{(t)}(z) - \mu zq^{(t-1)}(z) + \mathbf{y}$. We now consider distributed implementation of the iterative algorithms of the form (1),(2), which is the setting of our main contribution.

B. Contribution: Cross-Iteration Coded Computing

Consider a master-worker framework (see Fig. 1), where the master node receives the input and distributes the input to P workers. The standard baseline approach, *BaselineParallel*, to distributed implementation of the above algorithm is to perform n iterations in n rounds of communication between the master and the workers, as described in Algorithm 1. In BaselineParallel, we split matrices $\mathbf{A} = \begin{bmatrix}\mathbf{A}_1\\ \mathbf{A}_2\\ \vdots\\ \mathbf{A}_P\end{bmatrix}$ where $\mathbf{A}_i$ are $N/P \times N$ matrices, and worker node i stores matrices $\mathbf{A}_i$ for $i = 1,2,\ldots,P$. The iterative algorithm (1),(2) is implemented in BaselineParallel with one round of communication between the master and the workers *per iteration*. Specifically, if the inputs $\mathbf{x}^{(0)},\mathbf{y}$ are sent in the beginning to the workers, then worker i sends $\mathbf{A}_i\mathbf{x}^{(0)}, \mathbf{A}_i\mathbf{y}$ for $i = 1,2,\ldots,P$; the master aggregates the results of P workers to obtain $\mathbf{A}\mathbf{x}^{(0)}, \mathbf{A}\mathbf{y}$. In the second round of communication, the master node sends to the workers, $\mathbf{A}\mathbf{x}^{(0)}, \mathbf{A}\mathbf{y}$; the workers multiply the received vectors with the matrices they store and send them back to the master node which then aggregates them to obtain $\mathbf{A}^2\mathbf{x}^{(0)}, \mathbf{A}^2\mathbf{y}$. Proceeding similarly, the master node obtains $\mathbf{A}^\ell\mathbf{x}^{(0)}, \mathbf{A}^\ell\mathbf{y}, \ell = 0,1,2,\ldots,n$ after n rounds of communication, and then obtains the output (1) by linearly combining the vectors based on (2).

Given that the baseline distributed approach requires n rounds of communication to compute n iterations, our work is motivated by the following question: *Can we perform n iterations of the iterative algorithm (1),(2) with fewer than n rounds of communication?*

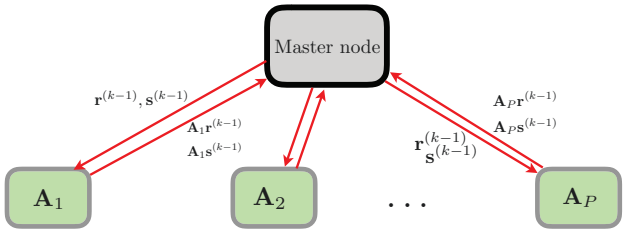

Fig. 1. The BaselineParallel algorithm in the master-worker architecture.

Algorithm 1 BaselineParallel$(\mathbf{A}, n, \mathbf{x}^{(0)}, \mathbf{y})$

1: **[A] Offline computations-Master node:** Split matrices $\mathbf{A}$ into P equal-dimension submatrices such that $\mathbf{A} = \begin{bmatrix}\mathbf{A}_1^T & \ldots & \mathbf{A}_P^T\end{bmatrix}^T$. Send submatrix $\mathbf{A}_i$ to the ith worker for $1 \le i \le P$.
2: **[B] Online computations**
3: Set $\mathbf{r}^{(0)} = \mathbf{x}^{(0)}$, $\mathbf{s}^{(0)} = \mathbf{y}$
4: **For** $k = 1 : n$ do
5: Send $\mathbf{r}^{(k-1)}, \mathbf{s}^{(k-1)}$ to each worker node
6:
7: **for** each worker $i \in \{1,2,\ldots,P\}$ **do** $\mathbf{r}_i^{(k)} = \mathbf{A}_i\mathbf{r}^{(k-1)}$ and $\mathbf{s}_i^{(k)} = \mathbf{A}_i\mathbf{s}^{(k-1)}$
8: send $\mathbf{x}_i^{(k)}$ and $\mathbf{b}_i^{(k)}$ to the master node.
9: Master node aggregates $\mathbf{r}_i^{(k)}$'s and $\mathbf{s}_i^{(k)}$'s to form $\mathbf{r}^{(k)} = \begin{bmatrix}\mathbf{r}_1^{(k)} & \ldots & \mathbf{r}_P^{(k)}\end{bmatrix}^T$ and $\mathbf{s}^{(k)} = \begin{bmatrix}\mathbf{s}_1^{(k)} & \ldots & \mathbf{s}_P^{(k)}\end{bmatrix}^T$.
10: **[C] Master Post-Processing:** Output $\sum_{\ell=0}^{n}\left(\alpha_i^{(n)}\mathbf{r}^{(i)} + \beta_i^{(n)}\mathbf{s}^{(i)}\right)$

Our main contribution, cross-iteration coded computing, answers the above question in affirmative through a non-trivial *distributed* approach to performing (1),(2). Note that a trivial approach to performing n iterations in only one round of communication is a *centralized* approach, where there is only one worker that stores $\mathbf{A}, \mathbf{Q}$, and performs the iterations (1),(2). However, note that this centralized approach would require each node storing $\mathbf{A}, \mathbf{Q}$ entirely, whereas in

the distributed approach that uses n rounds of communication, each node only stores a fraction $\frac{1}{P}$ of the matrices $\mathbf{A}, \mathbf{Q}$. Cross-iteration coded computing can be viewed as a trade-off between these two extremes. By incorporating redundancy in storage and computation, cross-iteration coded computing can perform several iterations of the algorithm with only one round of computation. In particular, by tuning the degree of storage/computation redundancy, cross-iteration coded computing can control the number of communication rounds performed.

C. Related Work

Most works on coded computing consider one-shot matrix operations or data shuffling [7]–[23]. For coding iterative algorithms, almost all existing work focuses on the optimization of a single iteration or weighted averaging in a single communication round [24]–[34]. Exceptions are [35], [36], which combine the intermediate results in two consecutive rounds to improve the performance of sparse coded computing. Our work is the first to code cross an arbitrary number of iterations, and leveraging this coding to reduce the number of communication rounds to one with a controlled increase in storage and computation costs.

The coded computing technique derived in this paper originates from "MatDot codes" [37], [38], which are storage-optimal codes for matrix multiplication, whose optimality was shown recently in [39]. MatDot codes, and their generalization, Poly-Dot codes, were recently applied to coded neural network training [40], [41]. Some contemporary works on polynomial-encoded matrix operations include [31], [42], and also, notably [43], which focuses on matrix operations with reduced communication costs. Compared to these works, our work applies to the general framework of multi-round iterative computation, with emphasis on techniques in distributed optimization. Perhaps most significantly, our technique can be useful in reducing communication costs, and thus speeding up the computation, even when no straggling is present. A technical report with some preliminary ideas of this paper, that specialized to the application of gradient descent (Example 1 above) is available at [44]. Our paper generalizes [44] significantly to include generic iterative algorithms of the form (1),(2), as well as introduces the concept of flexible recovery thresholds.

Next, in Section II, we describe our cross-iteration coded computing based algorithm and compare it with the baseline algorithm. In Section III, we describe the modification to the algorithm of Section II that can flexibly recover of the output based on the degree of straggling allowing for graceful performance degradation with increasing number of stragglers.

II. Cross-Iteration Coded Computing

We begin with some preliminary lemmas in Section II-A. Then we will describe an algorithm in Section II-B that will perform n iterations of iterative algorithm of (1),(2) in fewer than n rounds of communication.

A. Preliminary Lemmas

Consider an $N \times N$ matrix $\mathbf{A}$ that can be written as

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_2 & \dots & \mathbf{A}_{m-1} \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{A}}_0 \\ \overline{\mathbf{A}}_1 \\ \dots \\ \overline{\mathbf{A}}_{m-1} \end{bmatrix}$$

where $\mathbf{A}_i, i = 0, 1, \dots, m-1$ are a $N \times N/m$ matrices and $\overline{\mathbf{A}}_i$ are $N/m \times N$ matrices. Let $h_1(\zeta), h_2(\zeta)$ be respectively $N \times N/m$ and $N/m \times N$ matrices whose entries are polynomials in ζ as follows:

$$h_1(\zeta) = \sum_{i=0}^{m-1} \mathbf{A}_i \zeta^{m-i-1}, \quad h_2(\zeta) = \sum_{i=0}^{m-1} \overline{\mathbf{A}}_i \zeta^i.$$

Our first lemma describes a sequence of polynomials, where the i-th polynomial in the sequence has, as one of its coefficients, $\mathbf{A}^i$ for $i = 1, 2, \dots, n$. Furthermore, the sequence of polynomials is described via a recursive structure, which makes it amenable to iterative computation. This structure will be used in our algorithm in Section II-B.

Lemma 1. *For an even integer n, let $H^{(n)}(\zeta)$ be a matrix of polynomials defined recursively as follows:*

$$H^{(2)}(\zeta) = h_1(\zeta) h_2(\zeta) \tag{6}$$

$$H^{(n)}(\zeta) = h_1(\zeta^{m^{n/2-1}}) h_2(\zeta^{m^{n/2-1}}) H^{(n)}(\zeta) \tag{7}$$

Suppose we partition the $N \times N$ identity matrix $\mathbf{I}$ column-wise as $\mathbf{I} = \begin{bmatrix} \mathbf{I}_0 & \mathbf{I}_1 & \dots & \mathbf{I}_{m-1} \end{bmatrix}$, where $\mathbf{I}_j$ is a $N \times N/m$ matrix for $j = 1, 2, \dots, m$. For an odd integer $n > 1$, we define $H^{(n)}(\zeta)$ as

$$H^{(n)}(\zeta) = (\mathbf{I}_{m-1} + \mathbf{I}_{m-2}\zeta^{m^{\frac{n-1}{2}}} + \mathbf{I}_{m-3}\zeta^{2m^{\frac{n-1}{2}}} + \dots + \mathbf{I}_0 \zeta^{(m-1)m^{\frac{n-1}{2}}}) h_2(\zeta^{m^{\frac{n-1}{2}}}) H^{(n-1)}(\zeta) \tag{8}$$

Then, for $n > 1$, $\mathbf{A}^n$ is the co-efficient of $\zeta^{K_m(n)}$ in $H^{(n)}(\zeta)$ where

$$K_m(n) = \begin{cases} m^{n/2} - 1 & \text{if } n \text{ is even} \\ m^{\frac{n+1}{2}} - 1 & \text{if } n \text{ is odd} \end{cases} \tag{9}$$

The polynomial $H^{(n)}(\zeta)$ is a degree $2K_m(n)$ polynomial, and can therefore be interpolated with $2K_m(n)+1$ evaluation points. For reasons that will become clear in Section II-B, the number $2K_m(n)+1$ is referred to as the *recovery threshold* of our algorithm. The next lemma establishes a tangible connection between Lemma 1 and the iterative algorithm in (1),(2).

Lemma 2. *Let $p^{(n)}(z), q^{(n)}(z)$ be polynomials as described in (2). Then $p(\mathbf{A}), q(\mathbf{A})$ are respectively the coefficients of $\zeta^{K_m(n)}$ in*

$$T_1^{(n)}(\zeta) = \sum_{\ell=1}^{n} \alpha_\ell^{(n)} \zeta^{K_m(n) - K_m(\ell)} H^{(\ell)}(\zeta) + \alpha_0^{(n)} \zeta^{K_m(n)} \mathbf{I}$$

$$T_2^{(n)}(\zeta) = \sum_{\ell=1}^{n} \beta_\ell^{(n)} \zeta^{K_m(n)-K_m(\ell)} H^{(\ell)}(\zeta) + \beta_0^{(n)} \zeta^{K_m(n)} \mathbf{I}$$

where $\mathbf{I}$ *is the* $N \times N$ *identity matrix.*

Lemmas 1 and 2 are proved in the appendix.

B. Cross-Iteration Coded Computing Based Distributed Algorithm

We describe in Algorithm 2 the Cross-Iteration Coded Computing (CICC) algorithm, which performs n iterations of (1),(2) in one round of communication. In a system with P workers, the algorithm uses P distinct scalars $\zeta_1, \zeta_2, \ldots, \zeta_P$. The goal of the algorithm is to ensure that worker i computes and sends to the master node $T_1^{(n)}(\zeta_i)\mathbf{x}^{(0)}, T_2^{(n)}(\zeta_i)\mathbf{y}$, where $T_1^{(n)}(\zeta), T_2^{(n)}(\zeta)$ are as defined in Lemma 2. Since the degrees of $T_1^{(n)}, T_2^{(n)}$ are both $2K_m(n)$, the master node can interpolate $T_1^{(n)}(\zeta)\mathbf{x}^{(0)}, T_2^{(n)}(\zeta)\mathbf{y}$ from any $2K_m(n)+1$ worker nodes.

In the algorithm description, we use the following notation

$$h_k(\zeta) = \begin{cases} h_1(\zeta) & \text{if } k \text{ is even} \\ h_2(\zeta) & \text{if } k \text{ is odd} \end{cases}$$

As described in Line 14, the algorithm requires $P > 2K_m(n)$ workers to complete their iterations to compute the eventual output; for this reason, $2K_m(n)$ is referred to as a *recovery-threshold* for the algorithm. Note that the algorithm can tolerate $P - 2K_m(n)$ failed/straggling worker nodes.

Lines 3-5 in Algorithm 2 are designed based on equations (6),(7). Specifically, for even k, lines 3-5 at worker i effectively compute

$$h_1(\zeta_i^{m^{k/2}-1}) h_2(\zeta_i^{m^{k/2}-1}) h_1(\zeta_i^{m^{k/2}-2}) \ldots h_1(\zeta_i) h_2(\zeta_i) \mathbf{x}^{(0)},$$

which is equal to $H^{(k)}(\zeta_i)\mathbf{x}^{(0)}$. Thus for even k, $\mathbf{r}^{(k)}$ is equal to $H^{(k)}(\zeta_i)\mathbf{x}^{(0)}$. For odd k, line 11 ensures that $\mathbf{r}^{(k)}$ is also equal to $H^{(k)}(\zeta_i)\mathbf{x}^{(0)}$. Similarly, lines 3,6, 12 ensure that at worker i, $\mathbf{s}^{(k)}$ is equal to $H^{(k)}(\zeta_i)\mathbf{y}$ for $k = 1, 2, \ldots, n$. Line 13 is designed based on Lemma 2 and ensures that the master processing in Line 14 recovers $(p^{(n)}(\mathbf{A})\mathbf{x}^{(0)}, q^{(n)}(\mathbf{A})\mathbf{y})$.

C. Cost incurred by of the algorithm

In comparison with BaselineParallel algorithm which uses n rounds of communication, the CICC algorithm, Algorithm 2, uses only 1 round of communication. Furthermore, while BaselineParallel sends n vectors of length $N/P \times 1$ from each worker node to the master node, our CICC algorithm requires communication of only *one* vector of length $N \times 1$ to the master node. Thus CICC provides significant improvement in communication cost over the baseline algorithm in terms of number of rounds, and can also outperform the baseline algorithm in terms of the volume of data (number of bits) for certain parameters.

The CICC algorithm incurs an overhead in terms of computation and storage. In a system with P workers, the storage cost of BaselineParallel per worker is $O(N^2/P)$ whereas for CICC, it is $O(nN^2/m)$. In particular, if, like our baseline, we do not build any straggler tolerance in our algorithm, we have $P = 2m^{n/2} - 1$, so that $m = \left(\frac{P+1}{2}\right)^{2/n}$. So the storage cost of CICC is $O(\frac{nN^2}{(0.5(P+1))^{2/n}})$. Similarly, the computation cost of the baseline algorithm is $O(nN^2/P)$, whereas the computation cost of our algorithm is $O(\frac{nN^2}{(0.5(P+1))^{2/n}})$.

A simple generalization that intersperses the baseline algorithm with the CICC algorithm can be used to reduce communication costs. Specifically, for any ℓ that divides n, we can perform n iterations of (1),(2) with ℓ rounds of communication, by implementing CICC($\mathbf{A}, n/\ell, m, \tilde{\mathbf{x}^{(0)}}, \tilde{\mathbf{y}}$), where $\tilde{\mathbf{x}^{(0)}}$, $\tilde{\mathbf{y}}$, are based on the input in the previous round. In this version of the algorithm, the master node will obtain $\mathbf{A}^i\mathbf{x}^{(0)}, \mathbf{A}^i\mathbf{y}, i = 1, 2, \ldots, n$ via interpolation, and then use (1),(2) to linearly combine them into the appropriate output. For the sake of brevity, we do not describe this generalization in detail here (See [44], Sec. 5, for an example in the special case of gradient descent.). This generalization has computation and storage costs respectively equal to $O(\frac{nN^2}{(0.5(P+1))^{2/\ell}})$,

Algorithm 2 CICC($\mathbf{A}, n, m, \mathbf{x}^{(0)}, \mathbf{y}$)

1: **[A] Offline computations at Master node:** Let $\zeta_1, \zeta_2, \ldots, \zeta_P$ be distinct scalars. Send matrices $h_1(\zeta_i^{m^j}), h_2(\zeta_i^{m^j})$ for $j = 0, 1, 2, \ldots, \lfloor \frac{n-1}{2} \rfloor$, to node $i \in \{1, 2, \ldots, P\}$
2: **[B] Online computations at worker** i**:** Master sends $\mathbf{x}^{(0)}, \mathbf{y}$ to the ith worker for $i = 1, 2, \ldots P$.
3: $\tilde{\mathbf{r}}^{(0)} := \mathbf{x}^{(0)}, \tilde{\mathbf{s}}^{(0)} := \mathbf{y}$
4: **for** $k = 1 : n$ **do**
5: $\quad \tilde{\mathbf{r}}^{(k)} := h_k(\zeta_i^{m^{\lfloor (k-1)/2 \rfloor}}) \tilde{\mathbf{r}}^{(k-1)}$
6: $\quad \tilde{\mathbf{s}}^{(k)} := h_k(\zeta_i^{m^{\lfloor (k-1)/2 \rfloor}}) \tilde{\mathbf{s}}^{(k-1)}$
7: $\quad$ **if** k is even **then**
8: $\quad\quad \mathbf{r}_i^{(k)} := \tilde{\mathbf{r}}^{(k)}$
9: $\quad\quad \mathbf{s}_i^{(k)} := \tilde{\mathbf{s}}^{(k)}$
10: $\quad$ **else**
11: $\quad\quad \mathbf{r}_i^{(k)} := (\mathbf{I}_{m-1} + \mathbf{I}_{m-2}\zeta^{m^{\frac{k-1}{2}}} + \ldots + \mathbf{I}_0 \zeta^{(m-1)m^{\frac{k-1}{2}}}) \tilde{\mathbf{r}}^{(k)}$
12: $\quad\quad \mathbf{s}_i^{(k)} := (\mathbf{I}_{m-1} + \mathbf{I}_{m-2}\zeta^{m^{\frac{k-1}{2}}} + \ldots + \mathbf{I}_0 \zeta^{(m-1)m^{\frac{k-1}{2}}}) \tilde{\mathbf{s}}^{(k)}$
13: Send vector-pair $(\bar{\mathbf{r}}_i, \bar{\mathbf{s}}_i) = (\sum_{\ell=0}^{n} \zeta_i^{K_m(n)-K_m(\ell)} \alpha_\ell^{(n)} \mathbf{r}_i^{(\ell)}, \sum_{\ell=0}^{n} \zeta_i^{K_m(n)-K_m(\ell)} \beta_\ell^{(n)} \mathbf{s}_i^{(\ell)})$ to the master node, where $\mathbf{r}_i^{(0)} = \tilde{\mathbf{r}}^{(0)}, \mathbf{s}_i^{(0)} = \tilde{\mathbf{s}}^{(0)}$
14: **[C] Post-processing at Master node:** On receiving input $(\bar{\mathbf{r}}_i, \bar{\mathbf{s}}_i)$ from $2K_m(n)+1$ workers, using the evaluation points of these workers, compute the $N \times 1$ vector $\mathbf{R}$ by interpolating the vectors $\bar{\mathbf{r}}_i$ for a degree $2K_m(n)$ polynomial; that is, each component $\mathbf{R}$ is interpolated from the corresponding components of $\bar{\mathbf{r}}_i, i \in \{1, 2, \ldots, P\}$.
15: Similarly, compute $\mathbf{S}$ by interpolating $\bar{\mathbf{s}}_i$ for a degree $2K_m(n)$ polynomial with the evaluation points of the corresponding $2K_m(n)+1$ workers.
16: Output $\mathbf{R} + \mathbf{S}$

$O(\frac{nN^2}{\ell(0.5(P+1))^{2/\ell}})$, with a communication cost of ℓ rounds. This generalization can be used to tune computation/storage costs based on the desired communication cost.

Remark 1. In the gradient descent and accelerated gradient descent algorithms described in Section I-A, the vector $\mathbf{y}$ is of the form $\mathbf{Qb}$. In such settings, the vector $\mathbf{b}$ is the input and the matrix-vector multiplication $\mathbf{Qb}$ is a part of the online phase of the algorithm. The matrix-vector multiplication $\mathbf{Qb}$ can be readily incorporated both into BaselineParallel, and more importantly CICC by storing appropriate partitions/linear combinations of $\mathbf{Q}$ in the offline stage, and using $\mathbf{b}$ as the input. For the sake of brevity, we omit this modification in this paper (See [44] for an example).

III. Flexible Recovery Thresholds

In this section, we introduce the idea of *flexible recovery thresholds* for the iterative algorithm of (1),(2). We begin with the motivation in Section III-A. We then describe our algorithm in Section III-B.

A. Motivation

The idea of flexible recovery thresholds is relevant for distributed computing settings where stragglers are prevalent. Iterative algorithms of the form (1),(2) are usually associated with an error function that decays as the number of iterations increases. For instance, in linear inverse problems where gradient descent is used, the error is a measure of the distance between the output of the nth iteration and the optimal solution. For such scenarios, with a fixed recovery threshold, the error is small when the number of non-straggling workers is larger than the recovery threshold, however the error can be large if the number of non-straggling workers is smaller than the recovery threshold. The idea of flexible recovery thresholds allows for more graceful degradation of error with increasing number of stragglers.

We present our idea in the context of the CICC algorithm of Section II-B. We assume that there is *deadline*, T_{d_L}, after which workers sends their results to the master node. Note that different workers would perform a different number of iterations of CICC because of straggling and other sources of variability in computation time. With a fixed number of iterations n, and a corresponding fixed recovery threshold, the master declares a *failure* if the number of workers that completed n iterations is smaller than the recovery threshold.

To understand this more quantitatively, let the number of iterations completed by worker j by time t be denoted by $I(j,t)$. The *worker-profile* function, $\mathcal{J}(\ell,t)$ is the number of workers that have completed at least ℓ iterations in time t, that is $|\{k : I(k,t) \geq \ell\}|$. Now with a fixed number of iterations n, a fixed recovery threshold K, and a deadline T_{dl}, we refer to a run of an algorithm as a *failure* if its worker profile $\mathcal{J}(\ell,t)$ satisfies $\mathcal{J}(n,T_{dl}) < K$. In particular, for CICC, a failure occurs if $\mathcal{J}(n,T_{dl}) < 2K_m(n)+1 = 2m^{\lfloor n/2 \rfloor} - 1$. A run of the algorithm that has not failed is referred to as successful.

Note that the worker-profile can be different in different runs of the algorithm. See Fig.2 for visual depictions of two runs of CICC, one with a successful (non-failed) run, and another with a failed run. If the worker computation time can be statistically modeled, e.g. [13], then it would be possible to evaluate the probability of failure for CICC.

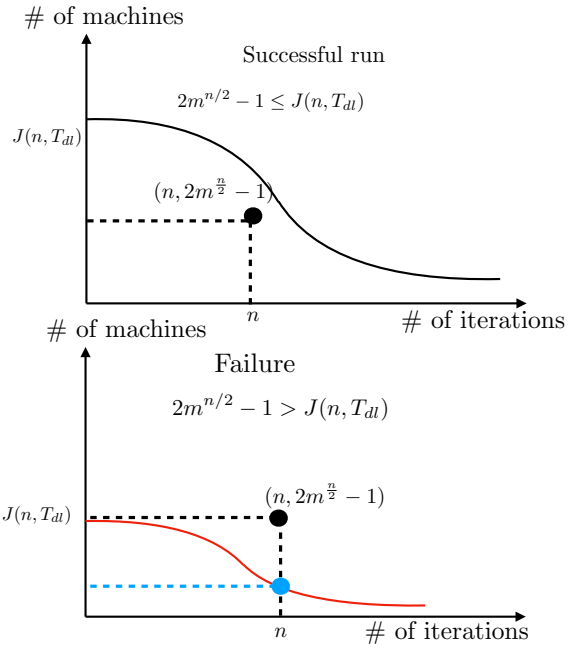

Fig. 2. Two runs of the CICC algorithm with fixed recovery threshold $K_m(n) = 2m^{\frac{n}{2}} - 1$. The run with the profile on the top is a successful run, whereas the profile on the bottom is a failure.

Flexible recovery thresholds reduces the chances of failure, by allowing the master node to choose the appropriate number of iterations in the post-processing stage, based on the number of stragglers. This is in contrast with fixed recovery thresholds algorithms, where the recovery threshold and the number of iterations are fixed in the offline pre-processing stage of the computation. For a given worker profile $\mathcal{J}(\ell,t)$, with a flexible recovery threshold, the master node can recreate the output of $n_{\mathcal{J}}$ iterations of CICC so long as $\mathcal{J}(n_{\mathcal{J}},T_{dl}) < 2K_m(n_{\mathcal{J}})+1$. This idea is visually depicted in Fig. 3. With a flexible recovery threshold, a *failure* does not occur so long as at least $2m-1$ workers complete at least one iteration, that is, $\mathcal{J}(1,T_{dl}) \geq 2K_m(1)+1 = 2m-1$. Note that the flexibility is particularly useful, when the algorithm has a "best-effort" requirement to get the lowest possible error with in deadline T_{dl}. Since the scenario with fixed recovery threshold is a special case of the flexible recovery threshold, in theory, flexible recovery threshold would also lower the probability of error if the worker output is modeled statistically.

B. Algorithm description

The offline and online phases of the algorithm with flexible recovery threshold are essentially identical to the CICC

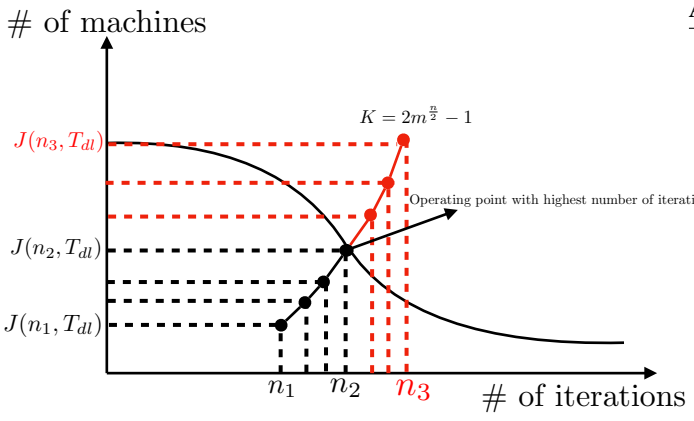

Fig. 3. A depiction of flexible recovery threshold. All the points colored black are feasible, and all the points colored red are in feasible. Unlike fixed recovery thresholds where the operating point is decided before the online phase, the algorithm can choose the operating point in the post-processing consensus phase.

algorithm, Algorithm 2 lines 1-12. However, in place of Line 13, we require a *consensus* phase, where the master/workers decide the number of iterations. Note that the number of iterations corresponding to profile $\mathcal{J}$ is the maximum number $n_{\mathcal{J}}$ satisfying

$$n_{\mathcal{J}} = \arg\max_{n} \mathcal{J}(n, T_{dl}) \geq K_m(n). \tag{10}$$

However, since $n_{\mathcal{J}}$ is not decided offline, and since each worker does not know how many iterations the other workers have performed, a consensus phase is required for the worker to include the output of the $n_{\mathcal{J}}$th iteration in its message to the master.

To achieve consensus, we require the workers to send the outputs of all their iterations to the master, and the master can reconstruct the worker profile, and then use the worker outputs to reconstruct the output of the algorithm. Note that this involves an increased communication cost as compared with the fixed-recovery threshold version, as each worker sends the output of all the iterations it completes, rather than only the output of the final iteration. This is depicted in Algorithm 3.

There is a second approach to achieving consensus. After completing their iterations, the workers send their number of iterations n_i to the master node. The master node determines the number of iterations $n_{\mathcal{J}}$ required based on (10) and sends this number to the workers. The workers respond to the master node with the outputs of that corresponding iteration, similar to Line 5 in Algorithm 2. We omit a formal description of this second approach for the sake of brevity. The question of which consensus approach is more likely to be useful in practice is a topic of future investigations.

IV. Conclusion

In this paper, we present a new approach for iterative distributed algorithms, such as gradient descent and accelerated gradient descent for quadratic loss functions, that can perform multiple iterations within a single round of communication. Cross-iteration coded computing thus saves on communication, one of the central bottlenecks in distributed computing systems, at an increased storage/computation cost. Because of the generic nature of the iterative algorithms considered in this paper, we anticipate that our work may be applicable beyond the applications explicitly listed in this paper. Our paper also demonstrates the idea of *flexible* recovery threshold that departs, in spirit from several previous coded computing works that use a fixed recovery threshold. An open question motivated by our work is whether the storage/computation overhead of our coding approach is optimal, even when restricted to simple linear schemes.

Complementary approaches that do not use coded computing involves ignoring stragglers [45], and communication-efficient distributed algorithms with weaker convergence guarantees (See [46]–[48] and references therein). The benefit of [45]–[48] in comparison with cross-iteration coded computing is that their storage/computation cost is similar to the baseline algorithm. A thematic direction of research motivated by our work is the idea of merging cross-iteration coded computing based and the ideas of [46]–[48] to reduce the storage cost of cross-iteration coded computing, but perhaps improve on the fidelity and convergence guarantees of [46]–[48].

Algorithm 3 CICC-flex($\mathbf{A}, T_{dl}, n_{max}, m, \mathbf{x}^{(0)}, \mathbf{y}$)

1: **[A] Offline computations at Master node:** Let $\zeta_1, \zeta_2, \ldots, \zeta_P$ be distinct scalars. Send matrices $h_1(\zeta_i^{m^j}), h_2(\zeta_i^{m^j})$ for $j = 0, 1, 2, \ldots, \lfloor \frac{n_{max}-1}{2} \rfloor$, to node $i \in \{1, 2, \ldots, P\}$
2: **[B] Online computations at worker i:**
3: Perform lines 1-12 in Algorithm 2 until time T_{dl}, or until n_{max} iterations, whichever completes first.
4: Denote the number of iterations performed as n_i.
5: Send vector-pairs $(n_i, \mathbf{r}_i^{(k)}, \mathbf{s}_i^{(k)}), k = 1, 2, \ldots, n_i$ to the master node.
6: **[C] Post-processing at Master node:**
7: On receiving inputs from the workers $(n_i, \mathbf{r}_i^{(k)}, \mathbf{s}_i^{(k)})$, construct worker profile $\mathcal{J}$ from $n_1, n_2, \ldots, n_P$ and use (10) to find the number of iterations $n_{\mathcal{J}}$.
8: For every worker i that has completed at least $n_{\mathcal{J}}$ iterations, compute $(\overline{\mathbf{r}}_i, \overline{\mathbf{s}}_i) = (\sum_{\ell=0}^{n_{\mathcal{J}}} \zeta_i^{K_m(n_{\mathcal{J}})-K_m(\ell)} \alpha_\ell \mathbf{r}_i^{(\ell)}, \sum_{\ell=0}^{n_{\mathcal{J}}} \zeta_i^{K_m(n_{\mathcal{J}})-K_m(\ell)} \beta_\ell \mathbf{s}_i^{(\ell)})$
9: Perform Lines 14,15 in Algorithm 2 for iteration index $n_{\mathcal{J}}$ with any of the $2K_m(n_{\mathcal{J}}) + 1$ workers that have completed $n_{\mathcal{J}}$ iterations, and then perform Line 16 to output.

Appendix A
Proof of Lemmas 1, 2

Proof of Lemma 1. We show this lemma for even n using induction. Later we handle the case of odd n. The co-efficient of ζ^{m-1} in $H^{(2)}(\zeta) = h_1(\zeta)h_2(\zeta)$ is $\sum_{i=0}^{m-1} \mathbf{A}_i \overline{\mathbf{A}}_i = \mathbf{A}^2$.

Assume, as an inductive hypothesis, that for some even $n \geq 2$, $\mathbf{A}^n$ is the co-efficient of $\zeta^{K_m(n)} = \zeta^{m^{n/2}-1}$ in $H^{(n)}(\zeta)$. We will show that $\mathbf{A}^{n+2}$ is the co-efficient of $\zeta^{K_m(n+2)} = \zeta^{m^{n/2}+1}$ in $H^{(n+2)} = h_1(\zeta^{m^{n/2}})h_2(\zeta^{m^{n/2}})H^{(n)}(\zeta)$.

Let

$$h_1(\zeta^{m^{n/2}})h_2(\zeta^{m^{n/2}}) = \sum_{i=0}^{2(m-1)m^{n/2}} \mathbf{C}_i\zeta^i,$$

$$H^{(n)}(\zeta^{m^{n/2+1}}) = \sum_{i=0}^{2(m^{n/2}-1)} \mathbf{D}_i\zeta^i.$$

Note that $\mathbf{C}_i = 0$ if $m^{n/2}$ does not divide i. Also note that $\mathbf{C}_{(m-1)m^{n/2}} = \mathbf{A}^2$ and, by inductive hypothesis, $\mathbf{D}_{m^{n/2}-1} = \mathbf{A}^n$. For convenience, we will assume that $d_\ell = 0$ for $\ell > 2(m^{n/2}-1)$. Thus the co-efficient of $\zeta^{K_m(n)} = \zeta^{m^{n/2+1}-1}$ in $H^{(n+2)}(\zeta)$ is

$$\sum_{\ell=0}^{m^{n/2+1}-1} \mathbf{C}_{m^{n/2+1}-1-\ell}\mathbf{D}_\ell$$

Because $\mathbf{C}_\ell = \mathbf{0}$ if $m^{n/2}$ does not divide ℓ, and because $\mathbf{C}_\ell = \mathbf{0}$ for $\ell > 2(m^{n/2}-1)$ this co-efficient is equal to $\mathbf{C}_{(m-1)m^2} = \mathbf{D}_{m^{n/2}-1} = \mathbf{A}^2\mathbf{A}^n = \mathbf{A}^{n+2}$. This completes the proof for even n.

Consider the case where n is odd. Let

$$h_2(\zeta^{m^{(n-1)/2}})H^{(n-1)}(\zeta) = \sum_{i=0}^{m^{(n-1)/2}+m^{(n+1)/2}-2} \mathbf{E}_i\zeta^i,$$

Since $n-1$ is even, from the above part of the proof, we know that $\mathbf{A}^{n-1}$ is the co-efficient of $\zeta^{K_m(n-1)}$ in $H^{(n-1)}(\zeta)$. Therefore, in polynomial $h_2(\zeta^{m^{(n-1)/2}})H^{(n-1)}(\zeta)$, the $N/m \times N$ matrix $\overline{\mathbf{A}}_i\mathbf{A}^{n-1}$ the co-efficient of $\zeta^{K_m(n-1)+i(m^{(n-1)/2})} = \zeta^{(i+1)m^{(n-1)/2}-1}$ for $i = 0, 1, 2, \ldots, m-1$. That is $\mathbf{E}_{(i+1)m^{(n-1)/2}-1} = \overline{\mathbf{A}}^i\mathbf{A}^{(n-1)}$. The co-efficient of $\zeta^{K_m(n)} = \zeta^{m^{(n+1)/2}-1}$ in polynomial

$$H^{(n)}(\zeta) = (\mathbf{I}_{m-1} + \mathbf{I}_{m-2}\zeta^{m^{\frac{n-1}{2}}} + \mathbf{I}_{m-3}\zeta^{2m^{\frac{n-1}{2}}} + \ldots + \mathbf{I}_0\zeta^{(m-1)m^{\frac{n-1}{2}}})h_2(\zeta^{m^{\frac{n-1}{2}}})H^{(n-1)}(\zeta)$$

is equal to

$$\sum_{\ell=0}^{m-1} \mathbf{I}_{m-\ell-1}\mathbf{E}_{m^{(n+1)/2}-1-\ell m^{(n-1)/2}} = \sum_{\ell=0}^{m-1} \mathbf{I}_{m-\ell-1}\mathbf{E}_{m^{(n-1)/2}(m-\ell)-1},$$

which is equal to $1\sum_{\ell=0}^{m-1} \mathbf{I}_{m-\ell-1}\overline{\mathbf{A}}_{m-\ell-1}\mathbf{A}^{n-1}$, which is in turn equal to $\mathbf{A}.\mathbf{A}^{n-1} = \mathbf{A}^n$ as required. ■

Proof of Lemma 2. From Lemma 1, we infer that $\mathbf{A}^\ell$ is the co-efficient of $\zeta^{K_m(n)}$ in $\zeta^{K_m(n)-K_m(\ell)}H^{(\ell)}(\zeta)$. Therefore, the co-efficient of $\zeta^{K_m(n)}$ in $T_1^{(n)}$ is $\sum_{\ell=0}^{n} \alpha_\ell^{(n)}\mathbf{A}^\ell$ which is equal to $p(n)(\mathbf{A})$. The proof is similar for $T_2^{(n)}(\zeta)$. ■

ACKNOWLEDGMENTS

This work was supported by NSF grants CCF-1350314, CCF-1553248, CCF-1763657 and CNS-1702694.

REFERENCES

[1] J. Dean, "Software engineering advice from building large-scale distributed systems," *CS295 Lecture at Stanford University, July*, 2007.

[2] V. Smith, S. Forte, M. Chenxin, M. Takáč, M. I. Jordan, and M. Jaggi, "Cocoa: A general framework for communication-efficient distributed optimization," *Journal of Machine Learning Research*, vol. 18, p. 230, 2018.

[3] S. Shalev-Shwartz and T. Zhang, "Stochastic dual coordinate ascent methods for regularized loss minimization," *Journal of Machine Learning Research*, vol. 14, no. Feb, pp. 567–599, 2013.

[4] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, and M. I. Jordan, "Communication-efficient distributed dual coordinate ascent," in *Advances in neural information processing systems*, 2014, pp. 3068–3076.

[5] X. Zhang, M. M. Khalili, and M. Liu, "Improving the privacy and accuracy of admm-based distributed algorithms," *arXiv preprint arXiv:1806.02246*, 2018.

[6] S. Maleki, M. Musuvathi, and T. Mytkowicz, "Parallel stochastic gradient descent with sound combiners," *arXiv preprint arXiv:1705.08030*, 2017.

[7] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," in *IEEE International Symposium on Information Theory (ISIT)*, 2016, pp. 1143–1147.

[8] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *Advances In Neural Information Processing Systems (NIPS)*, 2016, pp. 2092–2100.

[9] R. Bitar, P. Parag, and S. El Rouayheb, "Minimizing latency for secure distributed computing," in *Information Theory (ISIT), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 2900–2904.

[10] S. Wang, J. Liu, and N. Shroff, "Coded sparse matrix multiplication," *arXiv preprint arXiv:1802.03430*, 2018.

[11] A. Mallick, M. Chaudhari, and G. Joshi, "Rateless Codes for Near-Perfect Load Balancing in Distributed Matrix-Vector Multiplication," *arXiv preprint arXiv:1804.10331*, 2018.

[12] H. Jeong, T. M. Low, and P. Grover, "Coded FFT and Its Communication Overhead," *Submitted*, 2018.

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

[14] F. Haddadpour and V. R. Cadambe, "Codes for distributed finite alphabet matrix-vector multiplication," in *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 1625–1629.

[15] H. Park, K. Lee, J.-y. Sohn, C. Suh, and J. Moon, "Hierarchical coding for distributed computing," *arXiv preprint arXiv:1801.04686*, 2018.

[16] N. Ferdinand and S. C. Draper, "Hierarchical coded computation," in *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 1620–1624.

[17] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, "Coded computation over heterogeneous clusters," in *IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2408–2412.

[18] S. Li, M. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A Fundamental Tradeoff Between Computation and Communication in Distributed Computing," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, 2018.

[19] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded mapreduce," in *Communication, Control, and Computing (Allerton)*, 2015, pp. 964–971.

[20] S. Kiani, N. Ferdinand, and S. C. Draper, "Exploitation of stragglers in coded computation," in *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 1988–1992.

[21] N. S. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*. IEEE, 2016, pp. 954–960.

[22] G. Suh, K. Lee, and C. Suh, "Matrix sparsification for coded matrix multiplication," in *Communication, Control, and Computing (Allerton)*, 2017, pp. 1271–1278.

[23] T. Baharav, K. Lee, O. Ocal, and K. Ramchandran, "Straggler-proofing massive-scale distributed matrix multiplication with d-dimensional product codes," 2018.

[24] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *International Conference on Machine Learning (ICML)*, 2017, pp. 3368–3376.

[25] N. Raviv, I. Tamo, R. Tandon, and A. G. Dimakis, "Gradient Coding from Cyclic MDS Codes and Expander Graphs," *arXiv preprint arXiv:1707.03858*, 2017.

[26] W. Halbawi, N. Azizan-Ruhi, F. Salehi, and B. Hassibi, "Improving Distributed Gradient Descent Using Reed-Solomon Codes," *arXiv preprint arXiv:1706.05436*, 2017.

[27] C. Karakus, Y. Sun, and S. Diggavi, "Encoded distributed optimization," in *IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2890–2894.

[28] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler Mitigation in Distributed Optimization through Data Encoding," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 5440–5448.

[29] Z. Charles, D. Papailiopoulos, and J. Ellenberg, "Approximate gradient coding via sparse random graphs," *arXiv preprint arXiv:1711.06771*, 2017.

[30] R. K. Maity, A. S. Rawat, and A. Mazumdar, "Robust gradient descent via moment encoding with ldpc codes," *arXiv preprint arXiv:1805.08327*, 2018.

[31] Q. Yu, N. Raviv, J. So, and A. S. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security and privacy," *arXiv preprint arXiv:1806.00939*, 2018.

[32] Y. Yang, P. Grover, and S. Kar, "Fault-tolerant distributed logistic regression using unreliable components," in *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*. IEEE, 2016, pp. 940–947.

[33] Y. Yang, P. Grover, and S. Kar, "Coded Distributed Computing for Inverse Problems," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 709–719.

[34] S. Li, S. M. M. Kalan, Q. Yu, M. Soltanolkotabi, and A. S. Avestimehr, "Polynomially coded regression: Optimal straggler mitigation via data encoding," *arXiv preprint arXiv:1805.09934*, 2018.

[35] Y. Yang, P. Grover, and S. Kar, "Coding for a single sparse inverse problem," in *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 1575–1579.

[36] Y. Yang, M. Chaudhari, P. Grover, and S. Kar, "Coded iterative computing using substitute decoding," *arXiv preprint arXiv:1805.06046*, 2018.

[37] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," in *Communication, Control, and Computing (Allerton)*, Oct 2017, pp. 1264–1270.

[38] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *arXiv preprint arXiv:1801.10292, submitted to Transactions on Information Theory*, 2018.

[39] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," *arXiv preprint arXiv:1801.07487*, 2018.

[40] S. Dutta, Z. Bai, T. M. Low, and P. Grover, "Codenet: Training Large Neural Networks in presence of Soft-Errors," 2018.

[41] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, "A Unified Coded Deep Neural Network Training Strategy based on Generalized PolyDot codes," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 1585–1589.

[42] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication," in *Advances In Neural Information Processing Systems (NIPS)*, 2017, pp. 4403–4413.

[43] M. Ye and E. Abbe, "Communication-computation efficient gradient coding," *arXiv preprint arXiv:1802.03475*, 2018.

[44] F. Haddadpour, Y. Yang, M. Chaudhari, V. R. Cadambe, and P. Grover, "Straggler-resilient and communication-efficient distributed iterative linear solver," *arXiv preprint arXiv:1806.06140*, 2018.

[45] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[46] J. D. Lee, Q. Lin, T. Ma, and T. Yang, "Distributed stochastic variance reduced gradient methods by sampling extra data with replacement," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 4404–4446, 2017.

[47] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[48] Y. Zhang, M. J. Wainwright, and J. C. Duchi, "Communication-efficient algorithms for statistical optimization," in *Advances in Neural Information Processing Systems*, 2012, pp. 1502–1510.