Gradient Coding: Avoiding Stragglers in Distributed Learning

Rashish Tandon [1] **Qi Lei** [2] **Alexandros G. Dimakis** [3] **Nikos Karampatziakis** [4]

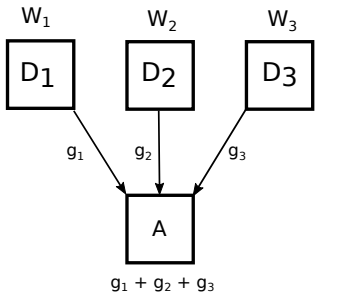
Abstract

We propose a novel coding theoretic framework for mitigating stragglers in distributed learning. We show how carefully replicating data blocks and coding across gradients can provide tolerance to failures and stragglers for synchronous Gradient Descent. We implement our schemes in python (using MPI) to run on Amazon EC2, and show how we compare against baseline approaches in running time and generalization error.

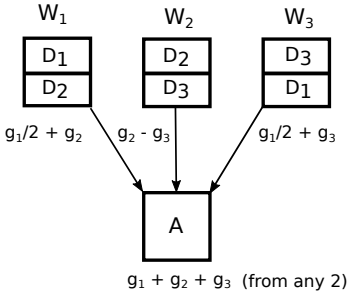
1. Introduction

We propose a novel coding theoretic framework for mitigating stragglers in distributed learning. The central idea can be seen through the simple example of Figure 1: Consider synchronous Gradient Descent (GD) on three workers (W_1,W_2,W_3). The baseline vanilla system is shown in the top figure and operates as follows: The three workers have different partitions of the labeled data stored locally (D_1,D_2,D_3) and all share the current model. Worker 1 computes the gradient of the model on examples in partition D_1, denoted by g_1. Similarly, Workers 2 and 3 compute g_2 and g_3. The three gradient vectors are then communicated to a central node (called the master/aggregator) A which computes the full gradient by summing these vectors $g_1+g_2+g_3$ and updates the model with a gradient step. The new model is then sent to the workers and the system moves to the next round (where the same examples or other labeled examples, say D_4,D_5,D_6, will be used in the same way).

The problem is that sometimes worker nodes can be stragglers (Li et al., 2014; Ho et al., 2013; Dean et al., 2012) *i.e.* delay significantly in computing and communicating gradient vectors to the master. This is especially pronounced for cheaper virtual machines in the cloud. For example on `t2.micro` machines on Amazon EC2, as can be seen in Figure 2: some machines can be $5\times$ slower in computing and communicating gradients compared to typical performance.



(a) Naive synchronous gradient descent



(b) Gradient coding: The vector $g_1+g_2+g_3$ is in the span of *any two* out of the vectors $g_1/2+g2$, g_2-g_3 and $g_1/2+g_3$.

Figure 1: The idea of Gradient Coding.

First, we discuss one way to resolve this problem if we replicate some data across machines by considering the placement in Fig.1 (b) but without coding. As can be seen, in Fig. 1 (b) each example is replicated two times using a specific placement policy. Each worker is assigned to compute two gradients on the two examples they have for this round. For example, W_1 will compute vectors g_1 and g_2. Now lets assume that W_3 is the straggler. If we use control messages, W_1, W_2 can notify the master A that they are done. Subsequently, if feedback is used, the master can ask W_1 to send g_1 and g_2 and W_2 to send g_3. These feedback control messages can be much smaller than the

[1]Department of Computer Science, University of Texas at Austin, Austin, TX, USA [2]Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX, USA [3]Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA [4]Microsoft, Seattle, WA, USA. Correspondence to: Rashish Tandon <rashish@cs.utexas.edu>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

actual gradient vectors but are still a system complication that can cause delays. However, feedback makes it possible for a centralized node to coordinate the workers, thereby avoiding straggler. One can also reduce network communication further by simply asking W_1 to send the sum of two gradient vectors $g_1 + g_2$ instead of sending both. The master can then create the global gradient on this batch by summing these two vectors. Unfortunately, which linear combination must be sent depends on who is the straggler: If W_2 was the straggler then W_1 should be sending g_2 and W_3 sending $g_1 + g_3$ so that their sum is the global gradient $g_1 + g_2 + g_3$.

In this paper we show that feedback and coordination is not necessary: every worker can send a *single linear combination of gradient vectors* without knowing who the straggler will be. The main coding theoretic question we investigate is how to design these linear combinations so *that any two* (or any fixed number generally) contain the $g_1 + g_2 + g_3$ vector in their span. In our example, in Fig. 1b, W_1 sends $\frac{1}{2}g_1 + g_2$, W_2 sends $g_2 - g_3$ and W_3 sends $\frac{1}{2}g_1 + g_3$. The reader can verify that A can obtain the vector $g_1 + g_2 + g_3$ from *any two out of these three vectors*. For instance, $g_1 + g_2 + g_3 = 2\left(\frac{1}{2}g_1 + g_2\right) - (g_2 - g_3)$. We call this idea *gradient coding*.

We consider this problem in the general setting of n machines and **any** s stragglers. We first establish a lower bound: to compute gradients on all the data in the presence of **any** s stragglers, each partition must be replicated $s + 1$ times across machines. We propose two placement and gradient coding schemes that match this optimal $s+1$ replication factor. We further consider a partial straggler setting, wherein we assume that a straggler can compute gradients at a fraction of the speed of others, and show how our scheme can be adapted to such scenarios. All proofs can be found in the supplementary material.

We also compare our scheme with the popular *ignoring the stragglers* approach (Chen et al., 2016): simply doing a gradient step when most workers are done. We see that while ignoring the stragglers is faster, this loses some data and which can hurt the generalization error. This can be especially pronounced in supervised learning with unbalanced labels or heavily unbalanced features since a few examples may contain critical, previously unseen information.

1.1. The Effects of Stragglers

In Figure 2, we show measurements on the time required for `t2.micro` Amazon EC2 instances to communicate gradients to a master machine (a `c3.8xlarge` instance). We observe that a few worker machines were incur a communication delay of up to $5x$ the typical behavior. Interestingly, throughout the timescale of our experiments (a few hours), the straggling behavior was consistent in the same machines.

We have also experimented extensively with other Amazon EC2 instances: Our finding is that cheaper instance types have significantly higher variability in performance. This is especially true for `t2` type instance which on AWS are described as having *Burstable Performance*. Fortunately, these machines have very low cost.

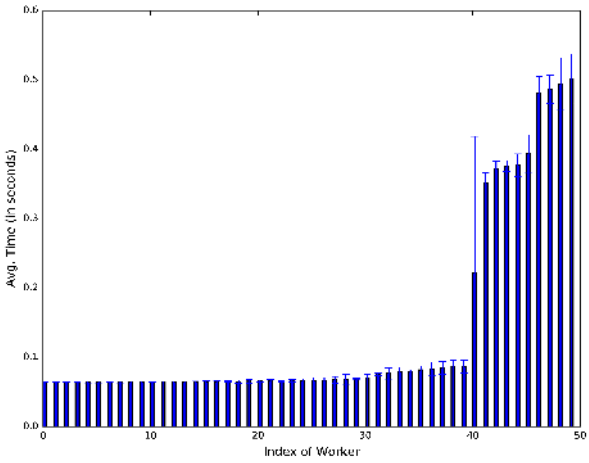


Figure 2: Average measured communication times for a vector of dimension $p = 500000$ using $n = 50$ `t2.micro` worker machines (and a `c3.8xlarge` master machine). Error bars indicate one standard deviation.

The choices of the number and type of workers used in training big models ultimately depends on total cost and time needed until deployment. The main message of this paper is that going for very low-cost instances and using coding to mitigate stragglers, may be a sensible choice for some learning problems.

1.2. Related Work

The slow machine problem is the Achilles heel of many distributed learning systems that run in modern cloud environments. Recognizing that, some recent work has advocated asynchronous approaches (Li et al., 2014; Ho et al., 2013; Mitliagkas et al., 2016) to learning. While asynchronous updates are a valid way to avoid slow machines, they do give up many other desirable properties, including faster convergence rates, amenability to analysis, and ease of reproducibility and debugging.

Attacking the straggling problem in synchronous machine learning algorithms has surprisingly not received much attention in the literature. There do exist general systems solutions such as speculative execution (Zaharia et al., 2008) but we believe that approaches tailored to machine learning can be vastly more efficient. In (Chen et al., 2016) the authors use synchronous minibatch SGD and request a small number of additional worker machines so that they have an adequate minibatch size even when some machines are slow. However, this approach does not handle well machines that are consistently slow and the data on those machines might never participate in training. In (Narayanamurthy et al., 2013) the authors describe an approach for dealing with

failed machines by approximating the loss function in the failed partitions with a linear approximation at the last iterate before they failed. Since the linear approximation is only valid at a small neighborhood of the model parameters, this approach can only work if failed data partitions are restored fairly quickly.

The work of (Lee et al., 2015) is the closest in spirit to our work, using coding theory and treating stragglers as erasures in the transmission of the computed results. However, we focus on codes for recovering the batch gradient of *any* loss function while (Lee et al., 2015) and the more recent work (Dutta et al., 2016) describe techniques for mitigating stragglers in two different distributed applications: data shuffling and matrix multiplication. We also mention (Li et al., 2016a), which investigates a generalized view of the coding ideas in (Lee et al., 2015). Further closely related work showed how coding can be used for distributed MapReduce and how to trade off communication and computation (Li et al., 2015), (Li et al., 2016b). All these prior works develop novel coding techniques, but do not code across gradient vectors in the way we are proposing in this paper.

2. Preliminaries

Given data $\mathbf{D} = \{(x_1, y_1), \ldots, (x_d, y_d)\}$, with each tuple $(x, y) \in \mathbb{R}^p \times \mathbb{R}$, several machine learning tasks aim to solve the following problem:

$$\beta^* = \arg\min_{\beta \in \mathbb{R}^p} \sum_{i=1}^{d} \ell(\beta; x_i, y_i) + \lambda R(\beta) \tag{1}$$

where $\ell(\cdot)$ is a task-specific loss function, and $R(\cdot)$ is a regularization function. Typically, this optimization problem can be solved using gradient-based approaches. Let $g := \sum_{i=1}^{d} \nabla \ell(\beta^{(t)}; x_i, y_i)$ be the gradient of the loss at the current model $\beta^{(t)}$. Then the updates to the model are of the form:

$$\beta^{(t+1)} = h_R\left(\beta^{(t)}, g\right) \tag{2}$$

where h_R is a gradient-based optimizer, which also depends on $R(\cdot)$. Several methods such as gradient descent, accelerated gradient, conditional gradient (Frank-Wolfe), proximal methods, LBFGS, and bundle methods fit in this framework. However, if the number of samples, d, is large, a computational bottleneck in the above update step is the computation of the gradient, g, whose computation can be distributed.

2.1. Notation

Throughout this paper, we let d denote the number of samples, n denote the number of workers, k denote the number of data partitions, and s denote the number of stragglers/failures. The n workers are denoted as $W_1, W_2, \ldots, W_n$. The partial gradients over k data partitions are denoted as $g_1, g_2, \ldots, g_k$. The i^{th} row of some matrices A or B is denoted as a_i or b_i respectively. For any vector $\mathbf{x} \in \mathbb{R}^n$, $\text{supp}(\mathbf{x})$ denotes its support *i.e.* $\text{supp}(\mathbf{x}) = \{i \,|\, \mathbf{x}_i \neq 0\}$, and $\|\mathbf{x}\|_0$ denotes its ℓ_0-norm i.e. the cardinality of the support. $\mathbf{1}_{p\times q}$ and $\mathbf{0}_{p\times q}$ denote all 1s and all 0s matrices respectively, with dimension $p \times q$. Finally, for any $r \in \mathbb{N}$, $[r]$ denotes the set $\{1, \ldots, r\}$.

2.2. The General Setup

We can generalize the scheme in Figure 1b to n workers and k data partitions by setting up a system of linear equations:

$$AB = \mathbf{1}_{f\times k} \tag{3}$$

where f denotes the number of combinations of surviving workers/non-stragglers, $\mathbf{1}_{f\times k}$ is the all 1s matrix of dimension $f \times k$, and we have matrices $A \in \mathbb{R}^{f\times n}$, $B \in \mathbb{R}^{n\times k}$.

We associate the i^{th} row of B, b_i, with the i^{th} worker, W_i. The support of b_i, $\text{supp}(b_i)$, corresponds to the data partitions that worker W_i has access to, and the entries of b_i encode a linear combination over their gradients that worker W_i transmits. Let $\bar{g} \in \mathbb{R}^{k\times d}$ be a matrix with each row being the partial gradient of a data partition i.e.

$$\bar{g} = [g_1, g_2, \ldots, g_k]^T.$$

Then, worker W_i transmits $b_i\bar{g}$. Note that to transmit $b_i\bar{g}$, W_i only needs to compute the partial gradients on the partitions in $\text{supp}(b_i)$. Now, each row of A is associated with a specific failure/straggler scenario, to which tolerance is desired. In particular, any row a_i, with support $\text{supp}(a_i)$, corresponds to the scenario where the worker indices in $\text{supp}(a_i)$ are alive/non-stragglers.

Also, by the construction in Eq. (3), we have:

$$a_iB\bar{g} = [1, 1, \ldots, 1]\bar{g} = \left(\sum_{j=1}^{k} g_j\right)^T \text{ and,} \tag{4}$$

$$a_iB\bar{g} = \sum_{k\in\text{supp}(a_i)} a_i(k)(b_k\bar{g}) \tag{5}$$

Thus, the entries of a_i encode a linear combination which, when taken over the transmitted gradients of the alive/non-straggler workers, $\{b_k\bar{g}\}_{k\in\text{supp}(a_i)}$, would yield the full gradient.

Going back to the example in Fig. 1b, the corresponding A and B matrices under the above generalization are:

$$A = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & -1 & 0 \end{pmatrix}, \text{ and } B = \begin{pmatrix} 1/2 & 1 & 0 \\ 0 & 1 & -1 \\ 1/2 & 0 & 1 \end{pmatrix} \tag{6}$$

with $f = 3, n = 3, k = 3$. It is easy to check that $AB = \mathbf{1}_{3\times 3}$. Also, since every row of A here has exactly one zero, we say that this scheme is robust to any one straggler.

In general, we shall seek schemes, through the construction of (A, B), which are robust to **any** s stragglers.

The rest of this paper is organized as follows. In Section 3 we provide two schemes applicable to any number of workers n, under the assumption that stragglers can be arbitrarily slow to the extent of total failure. In Section 4, we relax this assumption to the case of worker slowdown (with known slowdown factor), instead of failure, and show how our constructions can be appended to be more effective. Finally, in Section 5 we present results of empirical tests using our proposed distribution schemes on Amazon EC2.

3. Full Stragglers

In this section, we consider schemes robust to any s stragglers, given n workers (with $s < n$). We assume that any straggler is (what we call) a *full straggler i.e.* it can be arbitrarily slow to the extent of complete failure. We show how to construct the matrices A and B, with $AB = \mathbf{1}$, such that the scheme (A, B) is robust to **any** s full stragglers.

Consider any such scheme (A, B). Since every row of A represents a set of non-straggler workers, all possible sets over $[n]$ of size $(n - s)$ must be supports in the rows of A. Thus $f = \binom{n}{n-s} = \binom{n}{s}$ *i.e.* the total number of failure scenarios is the number of ways to choose s stragglers out of n workers. Now, since each row of A represents a linear span over some rows of B, and since we require $AB = \mathbf{1}$, this leads us to the following condition on B:

Condition 1 (B-Span)**.** *Consider any scheme (A, B) robust to **any** s stragglers, given n workers (with $s < n$). Then we require that for every subset $I \subseteq [n], |I| = n - s$:*

$$\mathbf{1}_{1\times k} \in \textit{span}\{b_i \,|\, i \in I\} \tag{7}$$

where span$\{\cdot\}$ is the span of vectors.

The B-Span condition above ensures that the all **1**s vector lies in the span of any $n - s$ rows of B. This is of course necessary. However, it is also sufficient. In particular, given a B satisfying Condition 1, we can construct A such that $AB = \mathbf{1}$, and A has the support structure discussed above. The construction of A is described in Algorithm 1 (in MATLAB syntax), and we have the following lemma.

Lemma 1. *Consider $B \in \mathbb{R}^{n\times k}$ satisfying Condition 1 for some $s < n$. Then, Algorithm 1, with input B and s, yields an $A \in \mathbb{R}^{\binom{n}{s}\times n}$ such that $AB = \mathbf{1}_{\binom{n}{s}\times n}$ and the scheme (A, B) is robust to any s full stragglers.*

Based on Lemma 1, to obtain a scheme (A, B) robust to **any** s stragglers, we only need to furnish a B satisfying Condition 1. A trivial B that works is $B = \mathbf{1}_{n\times k}$, the all ones matrix. However, this is wasteful since it implies that each worker gets all the partitions and computes the full gradient. Our goal is to construct B satisfying Condition 1 while also being as sparse as possible in each row. In this regard, we have the following theorem, which gives a lower bound on the number of non-zeros in any row of B.

Theorem 1 (Lower Bound on B's density)**.** *Consider any scheme (A, B) robust to **any** s stragglers, given n workers (with $s < n$) and k partitions. Then, if all rows of B have the same number of non-zeros, we must have: $\|b_i\|_0 \geq \frac{k}{n}(s+1)$ for any $i \in [n]$.*

Theorem 1 implies that any scheme (A, B) that assigns the same amount of data to all the workers must assign at least $\frac{s+1}{n}$ fraction of the data to each worker. Since this fraction is independent of k, for the remainder of this paper we shall assume that $k = n$ i.e. the number of partitions is the same as the number of workers. In this case, we want B to be a square matrix satisfying Condition 1, with each row having atleast $(s+1)$ non-zeros. In the sequel, we demonstrate two constructions for B which satisfy Condition 1 and achieve the density lower bound.

3.1. Fractional Repetition Scheme

In this section, we provide a construction for B that works by replicating the task done by a subset of the workers. We note that this construction is only applicable when the number of workers, n, is a multiple of $(s + 1)$, where s is the number of stragglers we seek tolerance to. In this case, the construction is as follows:

- We divide the n workers into $(s + 1)$ groups of size $(n/(s + 1))$.
- In each group, we divide all the data equally and disjointly, assigning $(s + 1)$ partitions to each worker
- All the groups are replicas of each other
- When finished computing, every worker transmits the sum of its partial gradients

Fig. 3 shows an instance of the above construction for $n = 6, s = 2$. A general description of B constructed in this way (denoted as B_{frac}) is shown in Eq. (9). Each group of workers in this scheme can be denoted by a block matrix

Algorithm 1 Algorithm to compute A

Input : B satisfying Condition 1, $s(< n)$
Output: A such that $AB = \mathbf{1}_{\binom{n}{s}\times n}$

$f = \text{binom}(n, s)$;
$A = \text{zeros}(f, n)$;
foreach $I \subseteq [n]$ *s.t.* $|I| = (n - s)$ **do**
  $a = \text{zeros}(1, k)$;
  $x = \text{ones}(1, k)/B(I, :)$;
  $a(I) = x$;
  $A = [A; a]$;
end

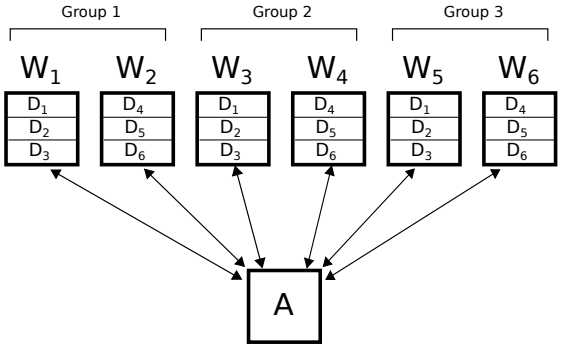


Figure 3: Fractional Repetition Scheme for $n = 6, s = 2$

$\overline{B}_{\text{block}}(n,s) \in \mathbb{R}^{\frac{n}{s+1} \times n}$. We define:

$$\overline{B}_{\text{block}}(n,s) = \begin{bmatrix} \mathbf{1}_{1\times(s+1)} & \mathbf{0}_{1\times(s+1)} & \cdots & \mathbf{0}_{1\times(s+1)} \\ \mathbf{0}_{1\times(s+1)} & \mathbf{1}_{1\times(s+1)} & \cdots & \mathbf{0}_{1\times(s+1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_{1\times(s+1)} & \mathbf{0}_{1\times(s+1)} & \cdots & \mathbf{1}_{1\times(s+1)} \end{bmatrix}_{\frac{n}{s+1}\times n} \tag{8}$$

Thus, the first worker in the group gets the first $(s+1)$ partitions, the second worker gets the second $(s+1)$ partitions, and so on. Then, B is simply $(s+1)$ replicated copies of $\overline{B}_{\text{block}}(n,s)$:

$$B = B_{frac} = \begin{bmatrix} \overline{B}^{(1)}_{\text{block}} \\ \overline{B}^{(2)}_{\text{block}} \\ \vdots \\ \overline{B}^{(s+1)}_{\text{block}} \end{bmatrix}_{n\times n} \tag{9}$$

where for each $t \in \{1,\ldots,s+1\}$, $\overline{B}^{(t)}_{\text{block}} = \overline{B}_{\text{block}}(n,s)$.

It is easy to see that this construction can yield robustness to **any** s stragglers. Since any particular partition of data is replicated over $(s+1)$ workers, any s stragglers would leave at least one non-straggler worker to process it. We have the following theorem.

Theorem 2. *Consider B_{frac} constructed as in Eq. (9), for a given number of workers n and stragglers $s(<n)$. Then, B_{frac} satisfies the B-Span condition (Condition 1). Consequently, the scheme (A, B_{frac}), with A constructed using Algorithm 1, is robust to any s stragglers.*

The construction of B_{frac} matches the density lower bound in Theorem 1 and, the above theorem shows that the scheme (A, B_{frac}), with A constructed from Algorithm 1, is robust to s stragglers.

3.2. Cyclic Repetition Scheme

In this section we provide an alternate construction for B which also matches the lower bound in Theorem 1 and satisfies Condition 1. However, in contrast to construction in the previous section, this construction does not require n to be divisible by $(s+1)$. Here, instead of assigning disjoint collections of partitions, we consider a cyclic assignment of $(s+1)$ partitions to the workers. We construct a $B = B_{cyc}$ with the following support structure:

Algorithm 2 Algorithm to construct $B = B_{cyc}$

Input : n, $s(<n)$
Output: $B \in \mathbb{R}^{n\times n}$ with $(s+1)$ non-zeros in each row

$H = \text{randn}(s, n)$;
$H(:,n) = -\text{sum}(H(:,1:n-1),2)$;
$B = \text{zeros}(n)$;
for $i = 1:n$ **do**
  $j = \text{mod}(i-1:s+i-1,n)+1$;
  $B(i,j) = [1; -H(:,j(2:s+1))\backslash H(:,j(1))]$;
end

$$\text{supp}(B_{cyc}) = \begin{bmatrix} \overbrace{\star & \star & \cdots & \star & \star}^{\text{s+1}} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \star & \star & \cdots & \star & \star & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \star & \star & \cdots & \star & \star \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \star & \cdots & \star & \star & 0 & 0 & \cdots & 0 & 0 & \star \end{bmatrix}_{n\times n} \tag{10}$$

where $\star$ indicates non-zero entries in B_{cyc}. So, the first row of B_{cyc} has its first $(s+1)$ entries assigned as non-zero. As we move down the rows, the positions of the $(s+1)$ non-zero entries shift one step to the right, and cycle around until the last row.

Given the support structure in Eq. 10, the actual non-zero entries must be carefully assigned in order to satisfy Condition 1. The basic idea is to pick every row of B_{cyc}, with its particular support, to lie in a suitable subspace S that contains the all ones vector $\mathbf{1}_{n\times 1}$. We consider a $(n-s)$ dimensional subspace, $S = \{x \in \mathbb{R}^n \mid Hx = 0,\ H \in \mathbb{R}^{s\times n}\}$ *i.e.* the null space of the matrix H, for some H satisfying $H\mathbf{1} = 0$. Now, to make the rows of B_{cyc} lie in S, we require that the null space of H must contain vectors with all the different supports in Eq. 10. This turns out to be equivalent to requiring that any s columns of H are linearly independent, and is also referred to as the MDS property in coding theory. We show that a random choice of H suffices for this, and we are able to construct a B_{cyc} with the support structure in Eq. 10. Moreover, for any $(n-s)$ rows of B_{cyc}, we show that their linear span also contains $\mathbf{1}_{n\times 1}$, thereby satisfying Condition 1. Algorithm 2 describes the construction of B_{cyc} (in MATLAB syntax) and, we have the following theorem.

Theorem 3. *Consider B_{cyc} constructed using the randomized construction in Algorithm 2, for a given number of*

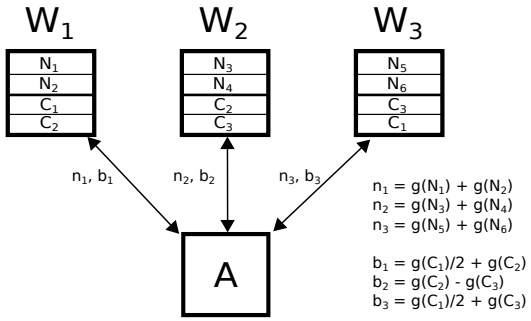


Figure 4: Scheme for Partial Stragglers, $n = 3, s = 1, \alpha = 2$. $g(\cdot)$ represents the partial gradient.

workers n and stragglers $s(< n)$. Then, with probability 1, B_{cyc} satisfies the B-Span condition (Condition 1). Consequently, the scheme (A, B_{cyc}), with A constructed using Algorithm 1, is robust to any s stragglers.

4. Partial Stragglers

In this section, we revisit our earlier assumption of *full stragglers*. Under a *full straggler* assumption, Theorem 1 shows that any non-straggler worker must incur an $(s + 1)$-factor overhead in computation, if we want to attain tolerance to any s stragglers. This may be prohibitively huge in many situations. One way to mitigate this is by allowing at least some work to be done also by the straggling workers. Therefore, in this section, we consider a more plausible scenario of slow workers, but assume a known slowdown factor. We say that a straggler is an α-*partial straggler* (with $\alpha > 1$) if it is at most α slower than any non-straggler. This means that if a non-straggler completes a task in time T, an α-*partial straggler* would require at most αT time to complete it. Now, we augment our previous schemes (in Section 3.1 and Section 3.2) to be robust to **any** s stragglers, assuming that any straggler is an α-*partial straggler*.

Note that our earlier constructions are still applicable: a scheme (A, B), with $B = B_{frac}$ or $B = B_{cyc}$, would still provide robustness to s partial stragglers. However, given that no machine is slower than a factor of α, a more efficient scheme is possible by exploiting at least some computation on every machine. Our basic idea is to couple our earlier schemes with a naive distribution scheme, but on different parts of the data. We split the data into a *naive* component, and a *coded* component. The key is to do the split such that whenever an α-partial straggler is done processing its *naive* partitions, a non-straggler would be done processing both its *naive* and *coded* partitions.

In general, for any (n, s, α), our two-stage scheme works as follows:

- We split the data $\mathbf{D}$ into $n + n\frac{s+1}{\alpha-1}$ equal-sized partitions — of which n partitions are *coded* components, and the rest are *naive* components
- Each worker gets $\frac{s+1}{\alpha-1}$ *naive* partitions, distributed disjointly.
- Each worker gets $(s + 1)$ *coded* partitions, distributed according to an (A, B) distribution scheme robust to s stragglers (e.g. with $B = B_{frac}$ or $B = B_{cyc}$)
- Any worker, W_i, first processes all its *naive* partitions and sends the sum of their gradients to the master. It then processes its *coded* partitions, and sends a linear combination, as per the (A, B) distribution scheme.

Note that each worker now has to send two partial gradients (instead of one, as in earlier schemes). However, a speedup gained in processing a smaller fraction of the data may mitigate this overhead in communication, since each non-straggler only has to process a $\frac{s+1}{n}\left(\frac{\alpha}{s+\alpha}\right)$ fraction of the data, as opposed to a $\frac{s+1}{n}$ fraction in *full straggler* schemes.

Fig. 4 illustrates our two-stage strategy for $n = 3, s = 1, \alpha = 2$. We see that each non-straggler gets $4/9 = 0.44$ fraction of the data, instead of a $2/3 = 0.67$ fraction (for e.g. in Fig 1b).

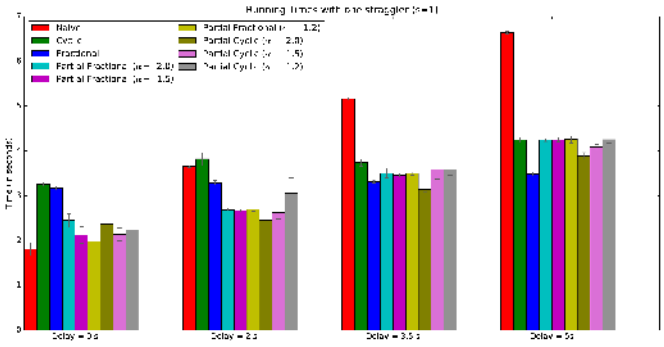
5. Experiments

In this section, we present experimental results on Amazon EC2, comparing our proposed gradient coding schemes with baseline approaches. We compare our approaches against: (1) the *naive* scheme, where the data is divided uniformly across all workers without replication and the master waits for all workers to send their gradients, and (2) the *ignoring s stragglers* scheme where the data is divided as in the naive scheme, however the master performs an update step after some $n - s$ workers have successfully sent their gradient.

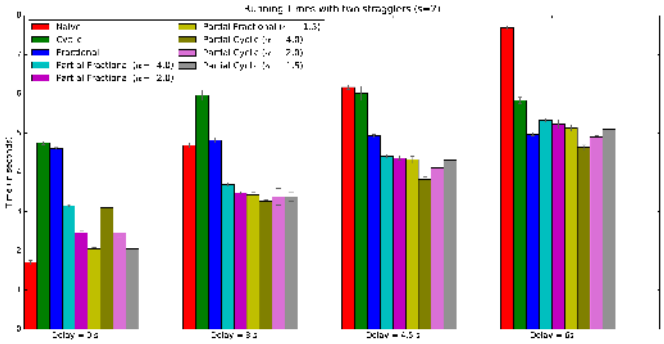
Experimental setup: We implemented all methods in python using MPI4py (Dalcin et al., 2011), an open source MPI implementation. Based on the method being considered, each worker loads a certain number of partitions of the data into memory before starting the iterations. In iteration t the master sends the latest model $\beta^{(t)}$ to all the workers (using `Isend()`). Each worker receives the model (using `Irecv()`) and starts a gradient computation. Once finished, it sends its gradient(s) back to the master. When sufficiently many workers have returned with their gradients, the master computes the overall gradient, performs a descent step, and moves on to the next iteration.

Our experiments were performed using two different worker instance types on Amazon EC2: `m1.small` and `t2.micro` — these are very small, very low-cost EC2 instances. We also observed that our system was often bottlenecked by the number of incoming connections *i.e.* all workers trying to talk to the master concurrently. For that reason, and to mitigate this additional overhead to some degree, we used a larger master instance of `c3.8xlarge` in our experiments.

We ran the various approaches to train logistic regression models, a well-understood convex problem that is widely used in practice. Moreover, Logistic regression models are often expanded by including interaction terms that are often



(a) $s = 1$ Straggler



(b) $s = 2$ Stragglers

Figure 5: Empirical running times on Amazon EC2 with $n = 12$ machines for $s = 1$ and $s = 2$ stragglers. In this experiment, the stragglers are artificially delayed while the other machines run at normal speed. We note that the partial straggler schemes have much lower data replication, for example with $\alpha = 1.2$ we need to only replicate approximately 10% of the data.

one-hot encoded for categorical features. This can lead to 100's of thousands of parameters (or more) in the trained models.

Results:

Artificial Dataset: In our first experiment, we solved a logistic regression problem using gradient descent, on a artificially generated dataset. We generated a dataset of $d = 554400$ samples $\mathbf{D} = \{(x_1, y_1), \ldots, (x_d, y_d)\}$, using the model $x \sim 0.5 \times \mathcal{N}(\mu_1, I) + 0.5 \times \mathcal{N}(\mu_2, I)$ (for random $\mu_1, \mu_2 \in \mathbb{R}^p$), and $y \sim Ber(p)$, with $p = 1/(\exp(2x^T\beta^*) + 1)$, where $\beta^* \in \mathbb{R}^p$ is the true regressor. In our experiments, we used a model dimension of $p = 100$, and chose β^* randomly.

In this experiment, we also artificially added delays to s random workers in each iteration (using `time.sleep()`). Figure 5 presents the results of our experiments with $s = 1$ and $s = 2$ stragglers, on a cluster of $n = 12$ `m1.small` machines. As expected, the baseline *naive* scheme that waits for the stragglers has poorer performance as the delay increases. The *Cyclic* and *Fractional* schemes were designed for one straggler in Figure 5a and for two stragglers 5b. Therefore, we expect that these two schemes would not be influenced at all by the delay of the stragglers (up to some variance due to implementation overheads). The *partial straggler* schemes were designed for various α. Recall that for partial straggler schemes, α denotes the *slowdown* factor.

Real Dataset: Next, we trained a logistic regression model on the Amazon Employee Access dataset from Kaggle [1]. We used $d = 26200$ training samples, and a model dimension of $p = 241915$ (after one-hot encoding with interaction terms). These experiments were run on $n = 10, 20, 30$ `t2.micro` instances on Amazon EC2.

In Figure 7 we show the Generalization AUC of our method (FracRep and CycRep) versus the popular approach of ignoring s stragglers. As can be seen, Gradient coding achieves significantly better generalization error. We emphasize that the results in figures 6 and 7 do not use any artificial straggling, only the natural delays introduced by the AWS cluster.

How is this stark difference possible? When stragglers are

[1] https://www.kaggle.com/c/amazon-employee-access-challenge

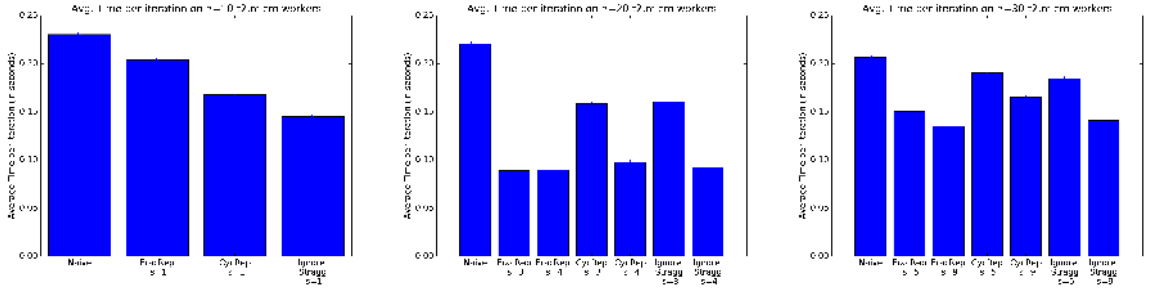


Figure 6: Avg. Time per iteration on Amazon Employee Access dataset.

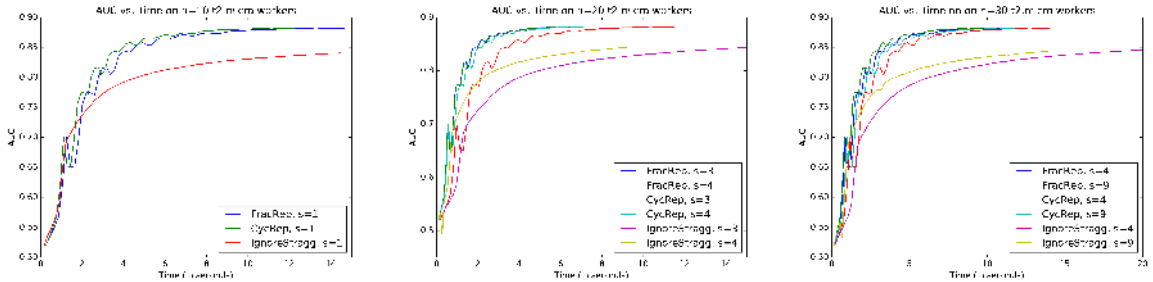


Figure 7: AUC vs Time on Amazon Employee Access dataset. The two proposed methods are FracRep and CycRep compared against the frequently used approach of Ignoring s stragglers. As can be seen, gradient coding achieves significantly better generalization error on a true holdout. Note that for Ignoring stragglers, we used a learning rate of $c_1/(t + c_2)$, with c_1 and c_2 chosen as optimal in a range. This is typical for SGD. For FracRep and CycRep, we used Nesterov's accelerated gradient descent with a constant learning rate (since these get the full gradient). Also, see discussion below in this regard.

ignored we are, at best, receiving a stochastic gradient (when random machines are straggling in each iteration). In this case the best we can do as an optimization algorithm is to run gradient descent as it is robust to noise. When using gradient coding however, we can retrieve the full gradient which gives us access to faster optimization algorithms. In Figure 7 we use Nesterov's Accelerated Gradient (NAG) but other optimizers are also applicable such as LBFGS. Though we do not present any empirical results here, we refer the reader to (Devolder et al., 2014) for a theoretical and empirical analysis of the effect of noisy gradients in NAG. The upshot is that ignoring stragglers cannot be combined with NAG because errors quickly accumulate and eventually cause the method to diverge.

Another advantage of using full gradients is that we can guarantee that we are training on the same distribution as the training set was drawn from. This is not true for the approach that ignores stragglers. If a particular machine is more likely to be a straggler, samples on that machine will likely be underrepresented in the final model, unless particular countermeasures are deployed. There may even be inherent reasons why a particular sample will systematically be excluded when we ignore stragglers. For example, in structured models such as linear-chain CRFs, the computation of the gradient is proportional to the length of the sequence. Therefore, extraordinarily long examples can be ignored very frequently.

6. Conclusion

In this paper, we have experimented with various *gradient coding* ideas on Amazon EC2 instances. This is a complex trade-off space between model sizes, number of samples, worker configurations, and number of workers. Our proposed schemes create computation overhead and keep communication the same.

The benefit of this additional computation is fault-tolerance: we are able to recover full gradients, even if s machines do not deliver their assigned work, or are slow in doing so. Our partial straggler schemes provide fault tolerance while allowing all machines to do partial work. Another interesting open question here is approximate gradient coding: can we get a vector that *is close to the true gradient*, with lesser computation overheads ?

For several model-cluster configurations that we tested, communication was the bottleneck and hence the additional computation's effect on iteration times was negligible. This is the regime where gradient coding is most useful. However, this design space needs further exploration, that is also varying as different architectures change the parameter landscape. Overall, we believe that gradient coding is an interesting idea to add in the distributed learning arsenal.

References

Chen, J., Monga, R., Bengio, S., and Jozefowicz, R. Revisiting Distributed Synchronous SGD. *ArXiv e-prints*, April 2016.

Dalcin, Lisandro D., Paz, Rodrigo R., Kler, Pablo A., and Cosimo, Alejandro. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools.

Dean, Jeffrey, Corrado, Greg, Monga, Rajat, Chen, Kai, Devin, Matthieu, Mao, Mark, Ranzato, Marc'Aurelio, Senior, Andrew, Tucker, Paul, Yang, Ke, Le, Quoc V., and Ng, Andrew Y. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*. 2012.

Devolder, Olivier, Glineur, François, and Nesterov, Yurii. First-order methods of smooth convex optimization with inexact oracle. *Mathematical Programming*, 146(1-2): 37–75, 2014.

Dutta, Sanghamitra, Cadambe, Viveck, and Grover, Pulkit. Short-dot: Computing large linear transforms distributedly using coded short dot products. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 2100–2108. Curran Associates, Inc., 2016.

Ho, Qirong, Cipar, James, Cui, Henggang, Lee, Seunghak, Kim, Jin Kyu, Gibbons, Phillip B, Gibson, Garth A, Ganger, Greg, and Xing, Eric P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, 2013.

Lee, Kangwook, Lam, Maximilian, Pedarsani, Ramtin, Papailiopoulos, Dimitris S., and Ramchandran, Kannan. Speeding up distributed machine learning using codes. *CoRR*, abs/1512.02673, 2015. URL http://arxiv.org/abs/1512.02673.

Li, Mu, Andersen, David G, Smola, Alex J, and Yu, Kai. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pp. 19–27, 2014.

Li, S., Maddah-Ali, M. A., and Avestimehr, A. S. Coded mapreduce. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 964–971, Sept 2015. doi: 10.1109/ALLERTON.2015.7447112.

Li, Songze, Maddah-Ali, Mohammad Ali, and Avestimehr, Amir Salman. A unified coding framework for distributed computing with straggling servers. *CoRR*, abs/1609.01690, 2016a. URL http://arxiv.org/abs/1609.01690.

Li, Songze, Maddah-Ali, Mohammad Ali, Yu, Qian, and Avestimehr, Amir Salman. A fundamental tradeoff between computation and communication in distributed computing. *CoRR*, abs/1604.07086, 2016b. URL http://arxiv.org/abs/1604.07086.

Mitliagkas, Ioannis, Zhang, Ce, Hadjis, Stefan, and Ré, Christopher. Asynchrony begets momentum, with an application to deep learning. *CoRR*, abs/1605.09774, 2016. URL http://arxiv.org/abs/1605.09774.

Narayanamurthy, Shravan, Weimer, Markus, Mahajan, Dhruv, Condie, Tyson, Sellamanickam, Sundararajan, and Keerthi, S. Sathiya. Towards resource-elastic machine learning, 2013. URL http://research.microsoft.com/apps/pubs/default.aspx?id=217296.

Zaharia, Matei, Konwinski, Andy, Joseph, Anthony D, Katz, Randy H, and Stoica, Ion. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, pp. 7, 2008.

Acknowledgements: RT, QL and AD would like to acknowledge the support of NSF Grants CCF 1344364, 1407278, 1422549, 1618689 and ARO YIP W911NF-14-1-0258, in conducting this research. RT and AD also acknowledge support from the William Hartwig Fellowship. NK thanks Paul Mineiro for insightful discussions.