Gradient Coding : Avoiding Stragglers in Synchronous Gradient Descent

Rashish Tandon*, Qi Lei*, Alexandros Dimakis*, Nikos Karampatziakis†

* The University of Texas at Austin, † Microsoft

Overview

We propose a novel coding theoretic framework for mitigating stragglers in distributed learning. We show how to achieve tolerance to **failures/stragglers** for **synchronous Gradient-based methods** by:

- Replicating data blocks across machines
- Coding over the gradients transmitted

Introduction

Given a data set $D = \{(x_1, y_1), \ldots, (x_d, y_d)\}$, many ML tasks require solving:

$$\beta^* = \underset{\beta \in \mathbb{R}^p}{\text{argmin}} \sum_{i=1}^{d} \ell(\beta; x_i, y_i) \tag{1}$$

Gradient-based methods solve this via the update step:

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

- $g^{(t)} := \sum_{i=1}^{d} \nabla \ell(\beta^{(t)}; x_i, y_i)$, is the **gradient**
- Several methods e.g. gradient descent, accelerated gradient descent, conditional gradient, LBFGS, bundle methods etc. fit in this framework
- If d is large, computation of $g^{(t)}$ can be distributed
- **KEY PROBLEM:** Some workers in a synchronous distributed setup can be **slow** or even **fail**

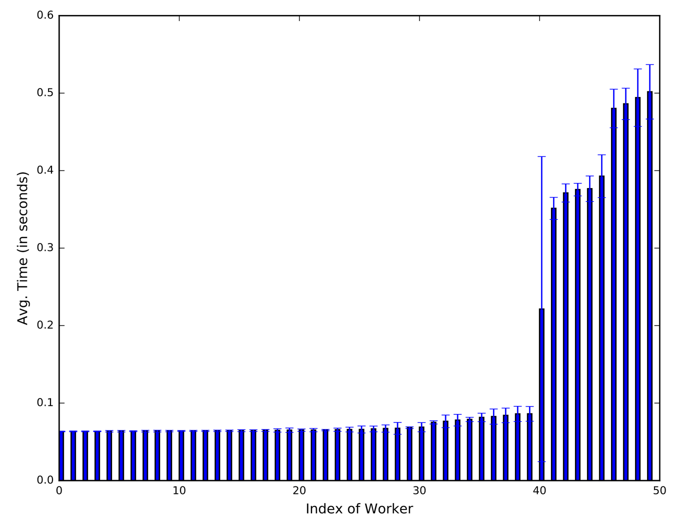
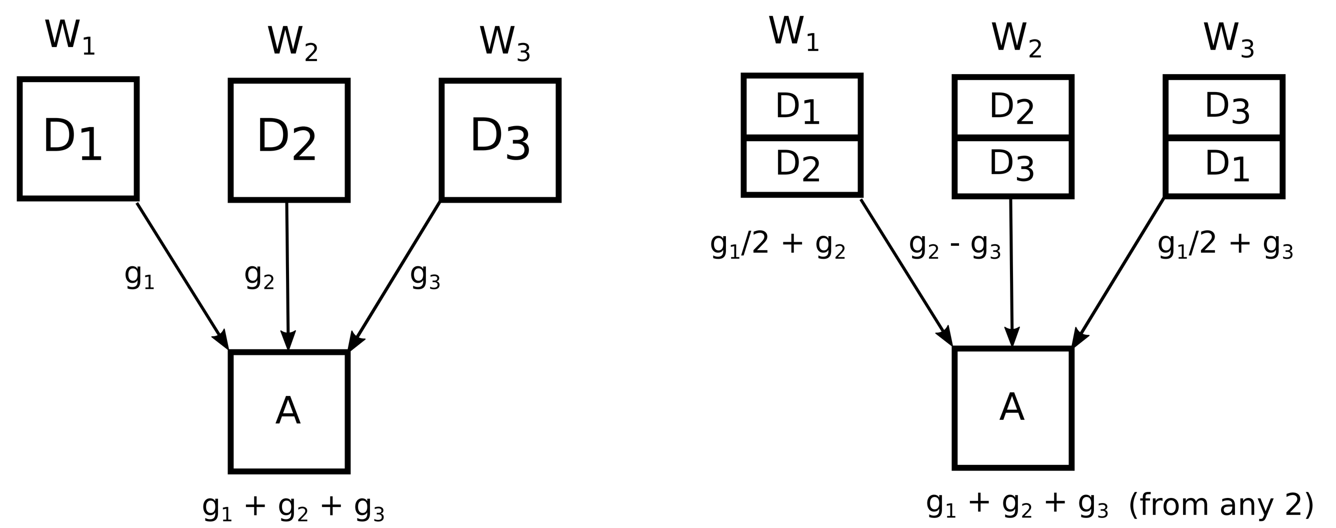


Figure: Average communication times (over 100 rounds), for a vector of dimension $p = 500000$ using $n = 50$ `t2.micro` workers, and a `c38x.large` master machine.



(a) Naive gradient descent (b) Coded gradient descent

Figure: Basic Idea of Gradient Coding

General Setup

- **Notation**: d samples, n workers denoted as $\{W_1, \ldots, W_n\}$, s stragglers (with $s < n$), Data partitions denoted as $\{D_1, \ldots, D_n\}$

We seek matrices $A \in \mathbb{R}^{f \times n}$, $B \in \mathbb{R}^{n \times n}$:

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

- f is number of combinations of non-straggler workers
- $\mathbf{1}_{f \times n}$ is the all 1s matrix of size $f \times n$
- B_i (i^{th} row of B) is *linear combination* over **partial gradients** computed by i^{th} worker
- A_i (i^{th} row of A) is *linear combination* over **workers** when *supp*(A_i) survive
- In the previous example:
  - $A = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & -1 & 0 \end{pmatrix}, B = \begin{pmatrix} 1/2 & 1 & 0 \\ 0 & 1 & -1 \\ 1/2 & 0 & 1 \end{pmatrix}$

Full Stragglers

- Stragglers are allowed to be **arbitrarily slow**, to the extent of complete failure
- Rows of A have all subsets over $[n]$ of size $(n-s)$ as supports $\left(f = \binom{n}{n-s}\right)$

We require that for any set $I \subseteq [n]$ s.t. $|I| = n - s$, B satisfies:

$$\mathbf{1}_{1 \times n} \in \text{span}\{B_i \,|\, i \in I\} \tag{4}$$

- In other words, any $n - s$ rows of B must contain the all 1s vector in their span
- Rows of B must also be *sparse*, to control computational overhead

Theorem (Lower Bound on B's density): If all rows of B have the same no. of non-zeros, then:

$$\|B_i\|_0 \geq (s+1) \text{ for all } i \in [n]$$

Fractional Repetition Scheme:

- Applicable when n is a multiple of $(s+1)$
- Partition data equally among $\frac{n}{(s+1)}$ workers
- Create $(s+1)$ replicas using other workers
- Every worker sends sum of its partial gradients

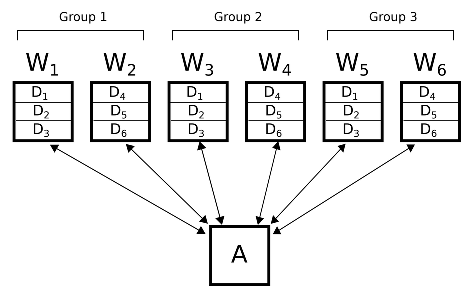


Figure: Fractional Repetition Scheme for $n = 6, s = 2$

Cyclic Repetition Scheme: B has the support structure

$$\text{supp}(B) = \begin{bmatrix} \overbrace{\star & \star & \cdots & \star & \star}^{s+1} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \star & \star & \cdots & \star & \star & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \star & \star & \cdots & \star & \star \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ \star & \cdots & \star & 0 & 0 & \cdots & 0 & \star \end{bmatrix}_{n \times n} \tag{5}$$

- Pick rows of B from a random subspace S (described as null-space of an $s \times n$ MDS matrix) and also containing $\mathbf{1}_{n \times 1}$
- Always possible for any n and s

Partial Stragglers

- Stragglers are allowed to be **at most α-times slower** than any non-straggler
- Key idea is to couple the naive scheme with the coding scheme
- Data is split into a *naive* component and *coded* component
- When a straggler finishes processing its *naive* components, a non-straggler finishes its *naive+coded* components

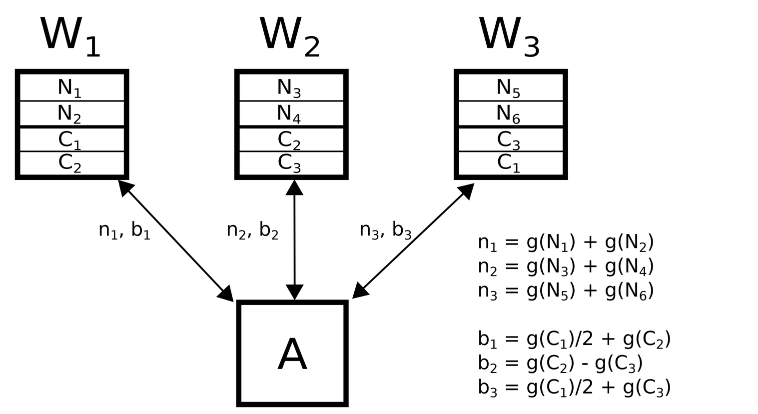
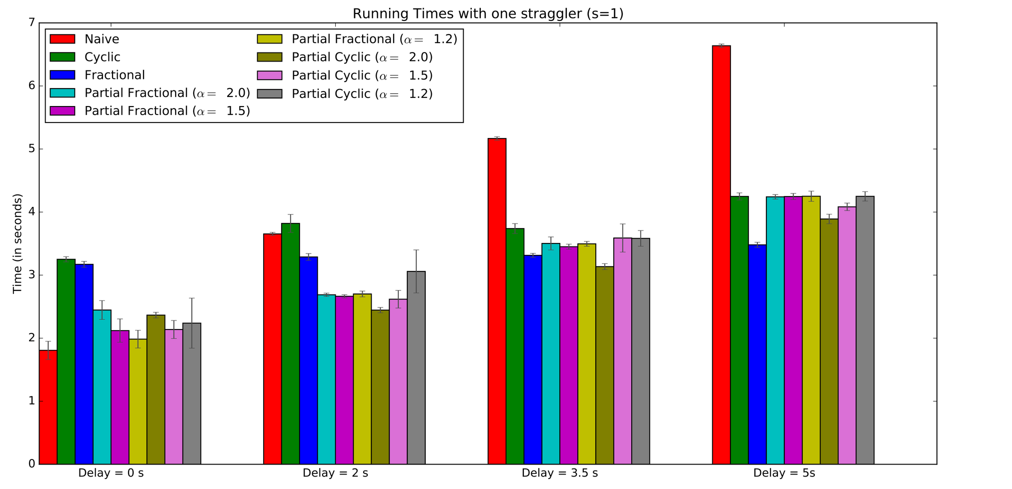


Figure: Scheme for Partial Stragglers, $n = 3, s = 1, \alpha = 2$. $g(\cdot)$ represents the partial gradient. Each worker gets 4/9 fraction of the data (as opposed to a 2/3 fraction in the earlier scheme).

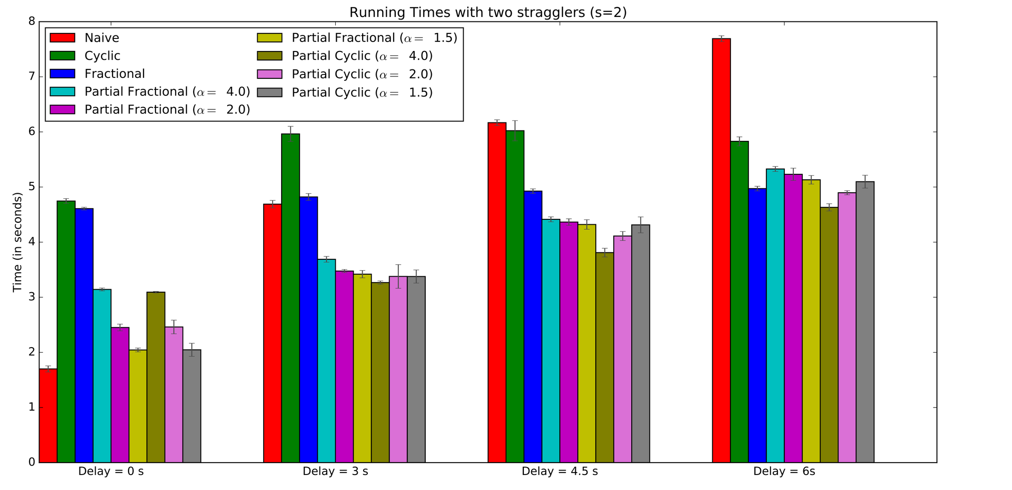
Experiments

Artificial Dataset:

- Generate a dataset with $d = 554400$ samples, $D = \{(x_1, y_1), \ldots, (x_d, y_d)\}$
- $x \sim \frac{1}{2}\mathcal{N}(\mu_1, I) + \frac{1}{2}\mathcal{N}(\mu_2, I)$ and $y \sim Ber(p)$, $p = 1/\left(\exp\left(2x^T\beta^*\right) + 1\right)$
- Model dimension $p = 100$, and μ_1, μ_2, β^* chosen randomly
- Train a logistic regression model



(a) One straggler



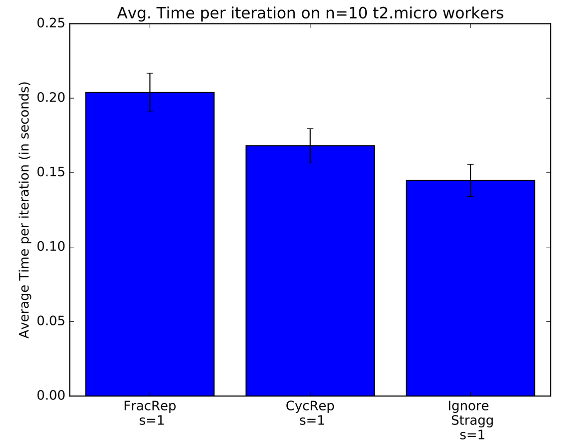
(b) Two stragglers

Figure: Empirical running time on EC2 with $n = 12$ `m1.small` machines, and one or two stragglers. Other machines run at normal speed while stragglers are artificially delayed. α is slowdown rate for partial schemes. We note that the partial straggler schemes for $\alpha = 1.2$ needs to only replicate approximately 10% of the blocks.

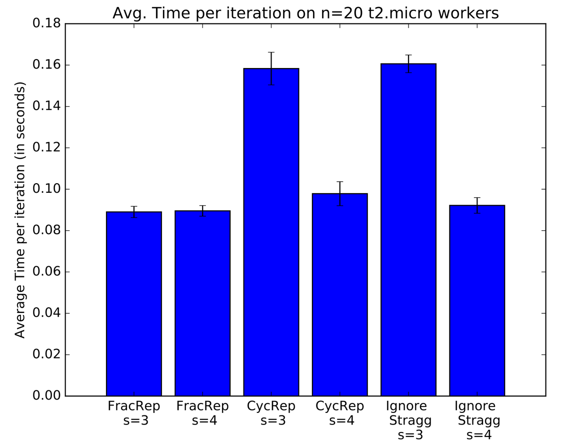
Experiments

Real Dataset:

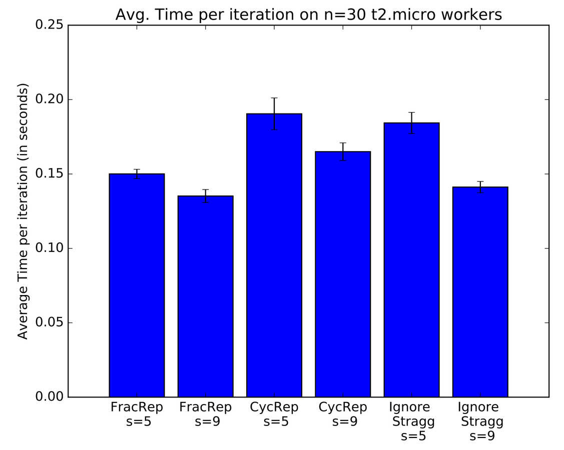
- Train a logistic regression model on Amazon Employee Access Dataset[1] ($d = 26200, p = 241915$)
- Comparison with *ignoring stragglers* approach *i.e.* data is distributed equally and we only wait for $(n - s)$ machines to finish



(a) $n = 10$

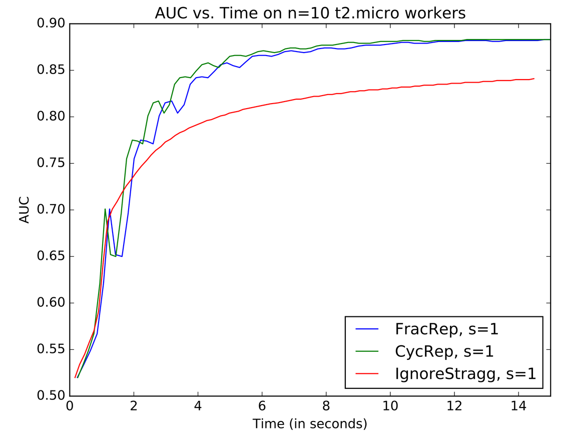


(b) $n = 20$

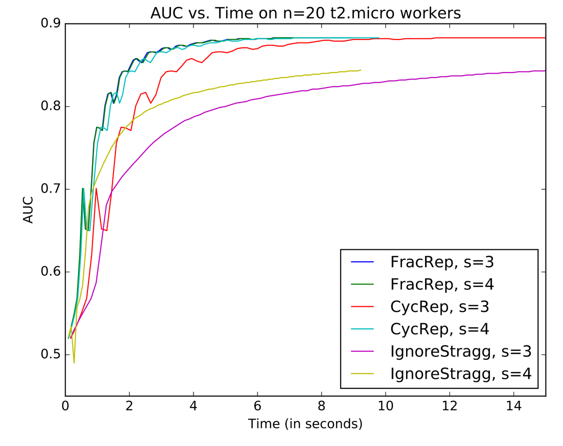


(c) $n = 30$

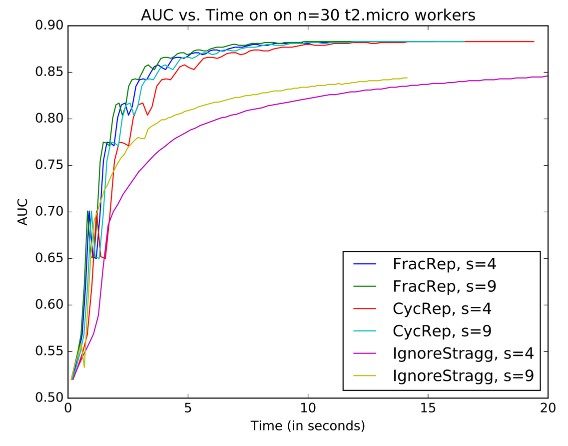
Figure: Avg. Time per iteration on Amazon Employee Access dataset, using `t2.micro` EC2 instances



(a) $n = 10$



(b) $n = 20$



(c) $n = 30$

Figure: AUC vs. Time on Amazon Employee Access dataset, using `t2.micro` EC2 instances

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