

A Sequential Approximation Framework for Coded Distributed Optimization

Jingge Zhu, Ye Pu, Vipul Gupta, Claire Tomlin, Kannan Ramchandran
EECS, University of California, Berkeley, CA, USA.

Email: {jingge.zhu, yepu, vipul_gupta, tomlin, kannanr}@eecs.berkeley.edu

Abstract—Building on the previous work of Lee *et al.* [2] and Ferdinand *et al.* [3] on coded computation, we propose a sequential approximation framework for solving optimization problems in a distributed manner. In a distributed computation system, latency caused by individual processors (“stragglers”) usually causes a significant delay in the overall process. The proposed method is powered by a sequential computation scheme, which is designed specifically for systems with stragglers. This scheme has the desirable property that the user is guaranteed to receive useful (approximate) computation results whenever a processor finishes its subtask, even in the presence of uncertain latency. In this paper, we give a coding theorem for sequentially computing matrix-vector multiplications, and the optimality of this coding scheme is also established. As an application of the results, we demonstrate solving optimization problems using a sequential approximation approach, which accelerates the algorithm in a distributed system with stragglers.

I. INTRODUCTION

Emerging applications from social networks and machine learning make distributed computation systems increasingly important for handling large-scale computation tasks. In this framework, a large computation task is divided into several smaller sub-tasks, each of which is dispatched to a different processor. The computation results are then aggregated and processed to produce the final result. A central challenge to this approach is how to handle uncertainty caused by “system noise” (see, e. g. [1]). One notable phenomenon is the “straggler” effect, namely, the latency of a single processor could cause a significant delay in the whole computational task. In existing distributed computation schemes, various straggler-detecting algorithms have been proposed to mitigate this problem. For example, Hadoop detects stragglers when executing the computation. When it detects a straggler, it runs a copy of the “straggled” task on a different processor. However, running many replications of subtasks turn out to be inefficient.

A novel approach to mitigating uncertainty is to add controlled redundancy in the distributed computation tasks. Lee *et al.* [2] proposed a coded computation framework for computing matrix-vector multiplications in a distributed system. By using maximum distance separable (MDS) codes to encode the matrix and distribute smaller computation tasks to different processors, they show that coded computation can

derive significant gains over naïve replication methods in terms of computation time. Based on the same idea, Ferdinand and Draper [3] proposed a refined coding scheme (called “anytime coding scheme”) where an approximation of the matrix-vector multiplication can be obtained in a timely fashion. We point out that the coded computation scheme has also been extended to study matrix multiplication problems [4], and is shown to be useful for reducing the communication overhead in distributed systems [5]. Furthermore, the idea of using codes in distributed systems has found various applications in machine learning problems, as shown in [6] [7] [8].

In this paper, we take a step further in studying how to tackle optimization problems using a coded computation approach. Building on the work of [2] and [3], we propose a *sequential approximation method* for solving optimization problems. The basic idea of this approach is that instead of directly solving the original problem, we solve a sequence of optimization problems (called *approximations*), whose solutions gracefully approach the solution of the original problem. These approximations need to be designed judiciously so that solving the approximate problems requires less computation time than solving the original problem. Consequently as we show in the sequel, in the presence of stragglers, the sequential approximation method typically takes less time to find the solution of the original problem. The saving on execution time is more significant if we only aim to find an approximate solution to the original problem. An attractive feature of the proposed method is that the processors in the distributed system are oblivious to different approximations, making it a user-centered design.

The driving mechanism for the proposed sequential approximation framework is a so-called *coded sequential computation scheme*, designed specifically for distributed computation systems with latency. It has the desirable property that the user is guaranteed to receive useful (approximate) computation results whenever a subset of processors finish their subtasks, even in the presence of uncertain latency. In this paper, we focus our study on a coding scheme for sequentially computing matrix-vector multiplication, which is a basic building block for most algorithms. We then show how to integrate our coded sequential computation scheme into the sequential approximation framework in order to accelerate the algorithm in the distributed computation system.

The work of Jingge Zhu was supported by the Swiss National Science Foundation under Project P2ELP2_165137. The work of Ye Pu was supported by the Swiss National Science Foundation under Project P2ELP2_165155.

II. THE SEQUENTIAL APPROXIMATION METHOD: AN OVERVIEW

We consider solving an optimization problem of the following form

$$\text{minimize } (1/2)\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{h}^T\mathbf{x} + g(\mathbf{x}) \quad (1)$$

where $\mathbf{H} \in \mathbb{R}^{m \times m}$ is positive (semi)-definite matrix and $g : \mathbb{R}^m \mapsto \mathbb{R}$ is a closed, proper and convex function. The formulation in (1) represents a large class of problems of interests. For example, choosing $g(\mathbf{x})$ to be the indicator function¹ $g(\mathbf{x}) = \mathbf{1}_{\mathbf{x} \in \mathbb{K}}$ converts problem (1) into a constrained optimization problem with a quadratic objective function, where $\mathbb{K} \subset \mathbb{R}^n$ is a convex set. Choosing $g(\mathbf{x})$ to be a norm of \mathbf{x} is also widely used in applications. For example, the choice $g(\mathbf{x}) = \gamma \|\mathbf{x}\|_1$ converts (1) to the lasso problem.

Alternating methods [] are efficient optimization methods used to solve problems of the form (1). The proximal gradient method [9] and ADMM are two examples of the alternating methods. For instance, the proximal gradient method update the variable \mathbf{x} as

$$\mathbf{x}^{k+1} = \text{prox}_{t^k g}(\mathbf{x}^k - t^k(\mathbf{H}\mathbf{x}^k + \mathbf{h})) \quad (2)$$

where \mathbf{x}^k and t^k denote the variable and the step size in the k -th iteration, respectively. The proximal operator prox is defined as

$$\text{prox}_{t g}(\mathbf{v}) := \text{argmin}_{\mathbf{x}}(g(\mathbf{x}) + (1/2t)\|\mathbf{x} - \mathbf{v}\|_2^2).$$

The update rule of ADMM is given as

$$\begin{aligned} \mathbf{x}^{k+1} &= \text{argmin}_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{h}^T\mathbf{x} + \mathbf{y}^{kT}(\mathbf{x} - \mathbf{z}^k) \\ &\quad + \frac{t^k}{2}\|\mathbf{x} - \mathbf{z}^k\|_2^2 \\ \mathbf{z}^{k+1} &= \text{argmin}_{\mathbf{z}} g(\mathbf{z}) + \mathbf{y}^{kT}(\mathbf{x}^{k+1} - \mathbf{z}) + \frac{t^k}{2}\|\mathbf{x}^{k+1} - \mathbf{z}\|_2^2 \\ \mathbf{y}^{k+1} &= \mathbf{y}^k + t^k(\mathbf{x}^{k+1} - \mathbf{z}^{k+1}) \end{aligned} \quad (3)$$

where \mathbf{y} denotes the Langrangian multiplier. To solve the first iteration in (3) using first order methods, we need to compute $\mathbf{H}\mathbf{x}^k$ for each steps.

From the above expressions, it can be argued that the matrix-vector multiplications $\mathbf{H}\mathbf{x}^k$ is (one of) the computationally most expensive operations² in this algorithm for each step k . With the focus on the matrix-vector multiplication, we will denote the update rule in (2) or (3) simply as $\mathbf{x}^{k+1} = f(\mathbf{H}\mathbf{x}^k)$ in the sequel.

In modern large-scale machine learning problems, the matrix \mathbf{H} can be very large so that computing the matrix multiplication (or even storing the matrix) in one processor is not feasible. In order to handle such large-scale problems, we turn to a distributed computation paradigm where the task of computing $\mathbf{H}\mathbf{x}^k$ is collaboratively accomplished by

several processors. As discussed in Introduction, the uncertainty (latency for example) of the individual processor could be detrimental to the distributed computation system and renders the distributed computation approach unusable. To alleviate this problem, previous works (e.g. [2] [3] [6]) have proposed coded computation schemes by adding redundancy in the computation tasks. We give a very simple example to illustrate the idea of coded computation. The matrix \mathbf{H} is vertically split into two smaller matrices $\mathbf{H}_1, \mathbf{H}_2$. We use three processors to store $\mathbf{H}_1, \mathbf{H}_2, \mathbf{H}_1 + \mathbf{H}_2$ separately, and each processor performs a smaller matrix-vector multiplication. It is easy to see that with *any* two of three multiplications $\mathbf{H}_1\mathbf{x}, \mathbf{H}_2\mathbf{x}, (\mathbf{H}_1 + \mathbf{H}_2)\mathbf{x}$, the user is able to recover $\mathbf{H}\mathbf{x}$. The same idea can be applied to a general setting with more users.

In this work, we take a step further to combine a coded sequential computation scheme with a modified algorithm. In particular, we propose the sequential approximation algorithm shown in Algorithm 1 for solving the problem in (1). In contrast to the original algorithm, Algorithm 1 executes a *sequence* of approximated problems (called ‘‘approximations’’). Each approximation is in the same form of the original problem, but with a different choice of the matrix $\mathbf{H}_{(r)}$. Notice that in order to obtain the correct solution in the end, the last approximation matrix $\mathbf{H}_{(R)}$ should be equal to \mathbf{H} . Algorithm 1 should possess the following two properties to be useful:

- 1) Executing the approximations $f(\mathbf{H}_{(r)}\mathbf{x}^k)$ should be faster than executing the original iteration $f(\mathbf{H}\mathbf{x}^k)$ in a distributed system.
- 2) The matrix $\mathbf{H}_{(r)}$ approaches the original matrix \mathbf{H} as r increases.

Property 1) ensures the proposed algorithm is faster than the original algorithm in the approximation phases, and Property 2) guarantees that Algorithm 1 eventually provides a solution which is close or identical to the solution of the original problem.

Algorithm 1 A sequential approximation for solving (1). The function $f(\cdot)$ represents the updating rule in (2) or (3)

Require: $\mathbf{H}_{(1)}, \dots, \mathbf{H}_{(R)}$ are R matrices which approximate \mathbf{H} with an increasing accuracy.

for $k = 1, \dots, T_1$ **do**
 $\mathbf{x}^{k+1} = f(\mathbf{H}_{(1)}\mathbf{x}^k)$
end for ▷ first approx. for T_1 iterations

for $k = T_1 + 1, \dots, T_1 + T_2$ **do**
 $\mathbf{x}^{k+1} = f(\mathbf{H}_{(2)}\mathbf{x}^k)$
end for ▷ second approx. for T_2 iterations

⋮
⋮
⋮

for $k = \sum_{r=1}^{R-1} T_r + 1, \dots, \sum_{r=1}^R T_r$ **do**
 $\mathbf{x}^{k+1} = f(\mathbf{H}_{(R)}\mathbf{x}^k)$
end for

An illustration of the sequential approximation approach is

¹The indicator function $\mathbf{1}_{\mathbf{x} \in \mathbb{K}}$ evaluates to 0 if $\mathbf{x} \in \mathbb{K}$ and to ∞ if $\mathbf{x} \notin \mathbb{K}$.

²Indeed, for example the proximal operator prox could be very simple for many problems of interests.

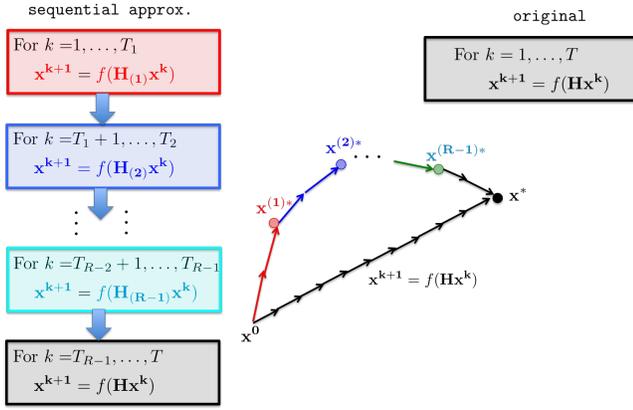


Fig. 1. An illustration of the sequential approximation approach. The black path on the bottom represents the path of \mathbf{x}^k when the variable is updated using the exact computation $\mathbf{H}\mathbf{x}$, and the colored “detour” represents the trajectory using sequential approximation. During the r -th approximation phase, the variable \mathbf{x}^k converges to the point $x^{(r)*}$, which is the optimal solution to the optimization problem (1) with \mathbf{H} replaced by $\mathbf{H}_{(r)}$. The sequential approximation approach could be faster in a distributed computation system if each iteration in the “detour” takes less computation time.

given in Fig. 1. The black trajectory on the bottom³ represents the path of \mathbf{x}^k when the variable is updated using the exact computation $\mathbf{H}\mathbf{x}$, and the colored “detour” represents the trajectory using the sequential approximation method.

Remark 1: It is natural to ask if the sequential approximation method in Algorithm 1 is already useful in a system without stragglers, where computing $\mathbf{H}_{(r)}\mathbf{x}$ takes the same amount of time as computing $\mathbf{H}\mathbf{x}$. A preliminary investigation suggests that it will depend on both the algorithm and the optimization problem (e.g. the condition number of the matrix). For certain problems, the sequential approximation approach can indeed provide a better convergence rate even for systems without stragglers. The results will be reported in our future work.

III. CODING FOR DISTRIBUTED SEQUENTIAL MATRIX-VECTOR MULTIPLICATION

A coded sequential computation scheme for the distributed system is the key mechanism behind the sequential approximation method. In this section, we formally introduce the coded sequential computation problem. It can be viewed as a general problem formulation of the anytime coding scheme studied in [3].

Consider a system with L processors where each processor performs a matrix-vector multiplication of the form $\mathbf{B}\mathbf{z}$. The matrix \mathbf{B} is of dimension $n \times m$ and \mathbf{z} is a m -length vector. Let $\mathbf{A}_1, \dots, \mathbf{A}_L$ be L matrices prescribed by the user where the matrix \mathbf{A}_i is of dimension $k_i \times m$. The goal is to compute the matrix-vector multiplications $\mathbf{A}_i\mathbf{z}, i = 1, \dots, L$ for a vector \mathbf{z} using these L processors. To perform the computation in a distributed manner, L matrices $\mathbf{B}_i \in \mathbb{R}^{n \times m}, i = 1, \dots, L$

³Notice that the actual trajectory of \mathbf{x}^k is not necessarily a straight line in \mathbb{R}^n . The plot is only an illustration.

are generated based on the given matrices $\mathbf{A}_i \in \mathbb{R}^{k_i \times m}, i = 1, \dots, L$, namely

$$\mathbf{B}_i = \mathcal{E}_i(\mathbf{A}_1, \dots, \mathbf{A}_L) \quad (4)$$

where \mathcal{E}_i denotes a mapping $\prod_j \mathbb{R}^{k_j \times m} \mapsto \mathbb{R}^{n \times m}$. The matrix \mathbf{B}_i is stored in the i -th processor. To compute the multiplication, a vector \mathbf{z} is given to all processors, and each processor returns the result $\mathbf{y}_i = \mathbf{B}_i\mathbf{z}$ to the user when it finishes the computation. The user then applies a decoder \mathcal{D} to obtain desired multiplications $\mathbf{A}_i\mathbf{z}$ using the received results \mathbf{y}_i . We would like to design our encoders \mathcal{E}_i and \mathcal{D} such that the system has the following property.

Property 1 (Sequential computation): With the computation results from any ℓ processors ($1 \leq \ell \leq L$), $\mathbf{y}^{(1)} := \mathbf{B}_{i_1}\mathbf{z}, \dots, \mathbf{y}^{(\ell)} := \mathbf{B}_{i_\ell}\mathbf{z}$ for some $i_1, \dots, i_\ell \in [L]$, the user can recover $\mathbf{A}_1\mathbf{z}, \dots, \mathbf{A}_\ell\mathbf{z}$ where $\mathbf{A}_i \in \mathbb{R}^{k_i \times m}$.

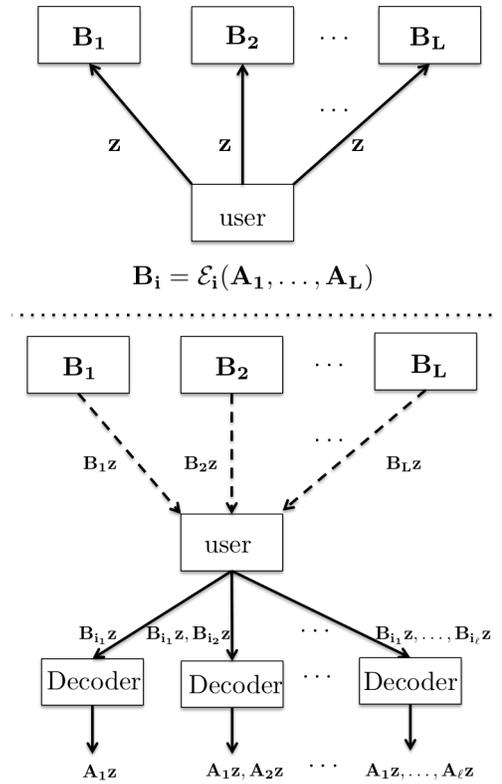


Fig. 2. An illustration for distributed sequential matrix-vector multiplication. L matrices $\mathbf{B}_1, \dots, \mathbf{B}_L$ are generated based on the prescribed matrices $\mathbf{A}_1, \dots, \mathbf{A}_L$. The user first sends a vector \mathbf{z} to all processors (top). Whenever a processor returns the computation result $\mathbf{B}_{i_\ell}\mathbf{z}$, the user can recover an additional matrix-vector multiplication $\mathbf{A}_\ell\mathbf{z}$ (bottom), $\ell = 1, \dots, L$.

Figure 2 gives an illustration of the sequential computation scheme of the distributed computation system. If a coding scheme satisfies the above sequential computation property, the corresponding value (k_1, \dots, k_L) is called a *feasible configuration*. The question is that given a distributed computation system with parameters $(L, n)^4$, what are the feasible

⁴As we shall see, the parameter m can be chosen arbitrarily.

configurations (possible values of k_1, \dots, k_L), and how do we design the encoders \mathcal{E}_i and the decoder \mathcal{D} for such a system.

This sequential computation scheme is useful for distributed computation systems with stragglers because it guarantees that any finished processor will provide useful results. Moreover, we could choose the matrices such that $\mathbf{A}_i \mathbf{z}$ is more crucial than $\mathbf{A}_j \mathbf{z}$ for our application if $i \leq j$, such that “more important” results are received earlier. As pointed out in [3], this coding scheme can also be viewed as an approximation method for computing the multiplication $\mathbf{A} \mathbf{z}$, where the accuracy increases gradually as more and more processors finish their tasks.

A. The coding scheme

In this section, we give a coding scheme for the sequential distributed matrix-vector multiplication problem. This scheme is a generalization of the MDS codes based scheme used in [2], and uses essentially the same idea as in [10] (multiple description coding) and [3].

Coding scheme: For each i , we divide the matrix \mathbf{A}_i vertically into at most $\lfloor k_i/i \rfloor + 1$ submatrices as follows

$$\begin{aligned} \mathbf{A}_i^{(1)} &:= \begin{pmatrix} \mathbf{A}_{i,1} \\ \vdots \\ \mathbf{A}_{i,i} \end{pmatrix}, \mathbf{A}_i^{(2)} := \begin{pmatrix} \mathbf{A}_{i,i+1} \\ \vdots \\ \mathbf{A}_{i,2i} \end{pmatrix}, \dots, \\ \mathbf{A}_i^{\lfloor k_i/i \rfloor} &:= \begin{pmatrix} \mathbf{A}_{i, \lfloor k_i/i \rfloor (i-1) + 1} \\ \vdots \\ \mathbf{A}_{i, \lfloor k_i/i \rfloor i} \end{pmatrix}, \\ \mathbf{A}_i^{\lfloor k_i/i \rfloor + 1} &:= \begin{pmatrix} \mathbf{A}_{i, \lfloor k_i/i \rfloor + 1} \\ \vdots \\ \mathbf{A}_{i, \lfloor k_i/i \rfloor + \text{mod}(k_i, i)} \end{pmatrix} \end{aligned}$$

where $\mathbf{A}_{i,j}$ denotes the j -th row of the matrix \mathbf{A}_i . In the case when i divides k_i , we do not have the last matrix $\mathbf{A}_i^{\lfloor k_i/i \rfloor + 1}$.

For each matrix $\mathbf{A}_i^{(j)} \in \mathbb{R}^{i \times m}$ where $j = 1, \dots, \lfloor k_i/i \rfloor$, we encode its rows to form a new matrix $\tilde{\mathbf{A}}_i^{(j)} \in \mathbb{R}^{L \times m}$. Particularly, we use a systematic MDS code, such that the first i rows of $\tilde{\mathbf{A}}_i^{(j)}$ is identical to $\mathbf{A}_i^{(j)}$, and the last $L - i$ rows are linear combinations of the rows of $\mathbf{A}_i^{(j)}$ (i.e., parity checks). If i does not divide k_i , the rows of the last matrix $\mathbf{A}_i^{\lfloor k_i/i \rfloor + 1} \in \mathbb{R}^{\text{mod}(k_i, i) \times m}$ is encoded with an $(L - i + \text{mod}(k_i, i), \text{mod}(k_i, i))$ MDS code into a new matrix $\tilde{\mathbf{A}}_i^{\lfloor k_i/i \rfloor + 1} \in \mathbb{R}^{(L - i + \text{mod}(k_i, i)) \times m}$.

The matrices \mathbf{B}_i are generated using the encoded matrices $\tilde{\mathbf{A}}_i^{(j)}$. More precisely, each matrix \mathbf{B}_i contains exactly one row of the matrix $\tilde{\mathbf{A}}_i^{(j)}$, for $j = 1, \dots, \lfloor k_i/i \rfloor$ and for all $i = 1, \dots, L$. If we have the extra matrix $\tilde{\mathbf{A}}_i^{\lfloor k_i/i \rfloor + 1}$ for certain i , its $L - i + \text{mod}(k_i, i)$ rows are distributed to arbitrary $L - i + \text{mod}(k_i, i)$ matrices \mathbf{B}_i .

Example: We apply the above coding scheme to a system with parameters $L = 4, n = 3$, and the configuration $(k_1 = 0, k_2 = 3, k_3 = 3, k_4 = 1)$. There are three matrices $\mathbf{A}_2 \in \mathbb{R}^{3 \times m}$, $\mathbf{A}_3 \in \mathbb{R}^{3 \times m}$ and $\mathbf{A}_4 \in \mathbb{R}^{1 \times m}$ to encode (\mathbf{A}_1 is zero

in this case). The proposed coding scheme generates matrices \mathbf{B}_i as follows:

$$\begin{aligned} \mathbf{B}_1 &= \begin{pmatrix} \mathbf{A}_{21} \\ \mathbf{A}_{23} \\ \mathbf{A}_{32} \end{pmatrix}, \mathbf{B}_2 = \begin{pmatrix} \mathbf{A}_{22} \\ \mathbf{A}_{23} \\ \mathbf{A}_{33} \end{pmatrix}, \mathbf{B}_3 = \begin{pmatrix} \mathbf{A}_{21} + \mathbf{A}_{22} \\ \mathbf{A}_{23} \\ \mathbf{A}_{31} + \mathbf{A}_{32} + \mathbf{A}_{33} \end{pmatrix}, \\ \mathbf{B}_4 &= \begin{pmatrix} \mathbf{A}_{21} + \mathbf{A}_{22} \\ \mathbf{A}_{31} \\ \mathbf{A}_4 \end{pmatrix}. \end{aligned}$$

It can be checked that we can recover $\mathbf{A}_2 \mathbf{z}$ if we have $\mathbf{B}_i \mathbf{z}, \mathbf{B}_j \mathbf{z}$ for any i, j , recover $\mathbf{A}_3 \mathbf{z}$ if we have $\mathbf{B}_i \mathbf{z}, \mathbf{B}_j \mathbf{z}, \mathbf{B}_k \mathbf{z}$ for any i, j, k and recover $\mathbf{A}_4 \mathbf{z}$ with all the computation results.

Theorem 1 (Coding scheme): Consider the distributed sequential matrix-vector multiplication problem with parameters (n, L) . The configuration (k_1, \dots, k_L) is feasible if it satisfies

$$\sum_{i=1}^L s_i \leq nL \quad (5)$$

where s_i is defined as

$$s_i := \begin{cases} \frac{k_i}{i} L & i \text{ divides } k_i \\ \lfloor \frac{k_i}{i} \rfloor \cdot L + L - i + \text{mod}(k_i, i) & \text{otherwise} \end{cases} \quad (6)$$

Proof: The proof is given in Appendix. ■

Remark 2: We point out that this result is a generalization of the coding scheme using a single MDS code proposed in [2], which can be seen as a special configuration with $(k_1 = 0, \dots, k_{\ell-1} = 0, k_\ell = \ell n, k_{\ell+1} = 0, \dots, k_L = 0)$ for some ℓ .

Remark 3 (Complexity of decoding): With the computation results from ℓ processors, the decoding process at the user is equivalent to solving a linear system of at most $n\ell$ unknowns, which does not depend on m (number of columns of the matrices). Hence this coding scheme is most beneficial for computing matrix-vector multiplications when the number of rows of matrices is very large. Moreover, using MDS codes with special structures (Reed-Solomon codes for example), the decoding process is often much simpler than solving a generic linear system.

We can show that if we restrict ourselves to linear coding schemes (i.e., the encoder in (4) is a linear function of $\mathbf{A}_1, \dots, \mathbf{A}_L$), then the coding method in Section III-A is the best possible. This result establishes the optimality of our coding scheme in the previous section.

Theorem 2 (Converse for linear schemes): Consider the distributed sequential matrix multiplication problem with parameters (n, L) . Under linear coding schemes, any feasible configuration (k_1, \dots, k_L) must satisfy the constraint (5).

Proof: The proof is given in Appendix. ■

IV. THE SEQUENTIAL APPROXIMATION METHOD: EXAMPLES

Equipped with the coded sequential computation scheme described in Section III, the sequential approximation method in Algorithm 1 can be implemented where the matrix-vector multiplication $\mathbf{H}_{(r)} \mathbf{x}^k$ is computed in a sequential manner. We

demonstrate this method by considering the Lasso problem in its standard form

$$\text{minimize } \frac{1}{2} \|\mathbf{F}\mathbf{x} - \mathbf{b}\|_2^2 + \gamma \|\mathbf{x}\|_1 \quad (7)$$

for a matrix $\mathbf{F} \in \mathbb{R}^{w \times m}$ and $\gamma \geq 0$. This corresponds to the optimization problem in (1) by identifying $\mathbf{H} = \mathbf{F}^T \mathbf{F}$, $\mathbf{h} = \mathbf{F}^T \mathbf{b}$ and $g(\mathbf{x}) = \gamma \|\mathbf{x}\|_1$. The corresponding proximal gradient method for this problem is given by

$$\mathbf{x}^{k+1} = S_\gamma(\mathbf{x}^k - t^k(\mathbf{F}^T \mathbf{F} \mathbf{x}^k - \mathbf{F}^T \mathbf{b})) \quad (8)$$

where $S_\gamma(x)$ is the *soft-thresholding operator* defined as

$$S_\gamma(x) = \begin{cases} x_i - \gamma & x_i \geq \gamma \\ 0 & |x_i| \leq \gamma \\ x_i + \gamma & x_i \leq -\gamma \end{cases}.$$

For this problem, the soft-thresholding operator is very simple, and the most computationally expensive step in the algorithm is the matrix-vector multiplication $\mathbf{F}^T \mathbf{F} \mathbf{x}^k$ for each step k .

A. Approximations

Instead of computing $\mathbf{F}^T \mathbf{F} \mathbf{x}^k$ at each step k , we use the proposed sequential distributed computation scheme. Similar to [2], we first focus on computing the term $\mathbf{F} \mathbf{x}^k$. Using Algorithm 1 to solve the above problem requires a specification of $\mathbf{F}_{(r)}$. A priori, $\mathbf{F}_{(r)}$ could be chosen in any way, as long as the multiplication $\mathbf{F}_{(r)} \mathbf{x}^k$ requires less computation than $\mathbf{F} \mathbf{x}^k$. Similar to [3], in this paper we choose $\mathbf{F}_{(r)}$ to be a low rank approximation of \mathbf{F} using singular value decomposition:

$$\mathbf{F} = \mathbf{U} \Sigma \mathbf{V}^T = \sum_{i=1}^d \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

where $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_d$ denote the singular values of \mathbf{F} with rank d . $\mathbf{u}_i \in \mathbb{R}^w$, $\mathbf{v}_i \in \mathbb{R}^m$ are the i -th column of \mathbf{U} and \mathbf{V} , respectively. In particular, we choose $\mathbf{F}_{(r)}$ as

$$\mathbf{F}_{(r)} := \sum_{i=1}^{\text{rank}(\mathbf{F}_{(r)})} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \quad (9)$$

for some $1 \leq \text{rank}(\mathbf{F}_{(r)}) \leq d$. Namely $\mathbf{F}_{(r)}$ captures the largest $\text{rank}(\mathbf{F}_{(r)})$ singular values of \mathbf{F} .

Define $\mathbf{A}_1 := [\mathbf{v}_1^T; \dots; \mathbf{v}_{k_1}^T] \in \mathbb{R}^{k_1 \times m}$, $\mathbf{A}_2 := [\mathbf{v}_{k_1+1}^T; \dots; \mathbf{v}_{k_1+k_2}^T] \in \mathbb{R}^{k_2 \times m}$, \dots , $\mathbf{A}_L := [\mathbf{v}_{d-k_L+1}^T; \dots; \mathbf{v}_d^T] \in \mathbb{R}^{k_L \times m}$ for a chosen configuration (k_1, \dots, k_L) which satisfies the condition (5). Using the coding scheme described in Section III-A, we generate L matrices $\mathbf{B}_1, \dots, \mathbf{B}_L \in \mathbb{R}^{n \times m}$ based on $\mathbf{A}_1, \dots, \mathbf{A}_L$ for the L processors.

Algorithm 2 Subroutine for computing $\mathbf{F}_{(r)} \mathbf{x}^k$: processor i

Require: \mathbf{B}_i is the encoded matrix stored in processor i

- 1: Receive an input vector \mathbf{x}^k
 - 2: Compute $\mathbf{y}_i := \mathbf{B}_i \mathbf{x}^k$
 - 3: Send \mathbf{y}_i back to the user
-

Algorithm 3 Subroutine for computing $\mathbf{F}_{(r)} \mathbf{x}^k$: user

- 1: Send a vector \mathbf{x}^k to all processors
 - 2: Wait until ℓ processors finish, where $\sum_{i=1}^{\ell} k_i \geq \text{rank}(\mathbf{F}_{(r)})$
 - 3: Decode $\mathbf{F}_{(r)} \mathbf{x}^k$ using $\mathbf{y}_{i_1}, \dots, \mathbf{y}_{i_\ell}$ from ℓ processors
-

When executing the algorithm, the vector \mathbf{x}^k is given to all processors at each time step k . The coding scheme guarantees that with the computation results $\mathbf{y}_i := \mathbf{B}_i \mathbf{x}^k$ from any ℓ processors ($1 \leq \ell \leq L$), the user can recover the multiplication result

$$\mathbf{t}_\ell := \begin{pmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_\ell \end{pmatrix} \mathbf{x}^k = \begin{pmatrix} \mathbf{v}_1^T \\ \vdots \\ \mathbf{v}_{h_\ell}^T \end{pmatrix} \mathbf{x}^k \in \mathbb{R}^{h_\ell} \quad (10)$$

where we define $h_\ell := \sum_{i=1}^{\ell} k_i$. An extra multiplication gives $\mathbf{F}_{(r)} \mathbf{x}^k$:

$$[\sigma_1 \mathbf{u}_1, \dots, \sigma_{h_\ell} \mathbf{u}_{h_\ell}] \mathbf{t}_\ell = \sum_{i=1}^{h_\ell} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \mathbf{x}^k = \mathbf{F}_{(r)} \mathbf{x}^k \quad (11)$$

where $\text{rank}(\mathbf{F}_{(r)}) = h_\ell$ in this case. We point out that in the lasso problem, the matrix \mathbf{F} in general has much more columns than rows, hence the above multiplication (11) takes less computation. Moreover, it can also be done using the distributed system in a similar way. By treating $\mathbf{F}_{(r)} \mathbf{x}^k$ as the vector to be multiplied, the user can distribute the multiplication $\mathbf{F}_{(r)}^T \mathbf{F}_{(r)} \mathbf{x}^k$ in the same way. Hence there are two computation steps for each iteration. We omit the details of the second step in this paper.

The subroutines for processors and for the user are given in Algorithm 2 and Algorithm 3, respectively. We point out that the subroutine for processors does not change for different approximation level (different $\mathbf{F}_{(r)}$), and only the user needs to adjust its procedure to adapt to different approximation levels.

B. Computation time

As mentioned in Section II, the reason for adopting the sequential approximation method is that it is faster to obtain a low rank approximation $\mathbf{F}_{(r)} \mathbf{x}^k$ than obtaining the exact answer $\mathbf{F} \mathbf{x}^k$ in a distributed system with stragglers. More precisely, let T_i denote the random computation time of processor i and let $T_{(\ell)}$ denote the ℓ -th order statistic, i.e.

$$T_{(\ell)} := \ell\text{-th smallest of } \{T_1, \dots, T_L\}.$$

The time for recovering the result $\mathbf{F}_{(r)} \mathbf{x}^k$ is given by $T_{(\ell)}$ where ℓ satisfies

$$\sum_{i=1}^{\ell} k_i \geq \text{rank}(\mathbf{F}_{(r)}),$$

while computing the exact answer $\mathbf{F} \mathbf{x}^k$ requires time $T_{(\ell')}$ where ℓ' satisfies

$$\sum_{i=1}^{\ell'} k_i \geq \text{rank}(\mathbf{F}).$$

Notice that we always have $\ell \leq \ell'$. In a distributed system with stragglers, $T_{(\ell')}$ could be significantly larger than $T_{(\ell)}$ if ℓ' is larger than ℓ .

In summary, although Algorithm 1 starts with “incorrect” iterations with approximation matrices $\mathbf{F}_{(r)}$, the computation time for those iterations are shorter than using the exact matrix \mathbf{F} . If we choose approximations $\mathbf{F}_{(r)}$ judiciously (approaching \mathbf{F} gradually), the variable \mathbf{x}^k will approach the optimal solution, but with a shorter computation time.

C. Choices of parameters

There are many free parameters to choose for the sequential approximation algorithm, including the approximation matrices $\mathbf{F}_{(r)}$, the number of different approximation levels R , and the number of iterations $T_r, r = 1, \dots, R$. They should be chosen in a way such that the algorithm can be implemented with the given system parameters (L, n) . In other words, if we choose $\mathbf{F}_{(r)}$ as in (9), there should exist a feasible configuration (k_1, \dots, k_L) which both satisfy condition (5) and the condition

$$\sum_{i=1}^{\ell(r)} k_i \geq \text{rank}(\mathbf{F}_{(r)}) \quad (12)$$

for every $r = 1, \dots, R$ with some $\ell(r) \in [1 : L]$ satisfying $\ell(r_1) \geq \ell(r_2)$ if $r_1 \geq r_2$. Recall that these choices will affect the running time of each iteration as shown in Subsection IV-B, hence we could optimize the overall execution time with respect to these parameters.

More importantly, choosing $\mathbf{F}_{(r)}$ as (9) is only one way to approximate the matrix \mathbf{F} . There may exist other choices of $\mathbf{F}_{(r)}$ which provide a better convergence performance. The problem of choosing good approximations will be addressed in the future work.

D. Numerical results

In this section, we present numerical results to demonstrate the sequential approximation method with the Lasso problem in (7). Specifically, we consider the proximal gradient algorithm in (8).

As a toy example, we assume that the distributed system has 4 processors where each processor can compute a matrix-vector multiplication \mathbf{Bz} with $\mathbf{B} \in \mathbb{R}^{10 \times m}$, namely $(L = 4, n = 10)$. The \mathbf{F} matrix in the Lasso problem has dimension $38 \times m$ with rank 38 and $m = 500$. In our simulations, the matrix \mathbf{F} is chosen randomly, and the regularization coefficient γ is chosen to be 5. The computation time of each processor is assumed to have an exponential distribution with the density function $p(x) = \lambda \exp(-\lambda x)$ with the choice $\lambda = 1$.

Example 1: We use two approximation levels where $\mathbf{F}_{(1)}, \mathbf{F}_{(2)}$ are chosen according to (9) with $\text{rank}(\mathbf{F}_{(1)}) = 6$ and $\text{rank}(\mathbf{F}_{(2)}) = 38$ (i.e., $\mathbf{F}_{(2)} = \mathbf{F}$). It can be checked straightforwardly this particular choice can be implemented with a feasible configuration $(k_1 = k_2 = 0, k_3 = 6, k_4 = 32)$. With this choice, the matrix-vector multiplication $\mathbf{F}_{(1)}\mathbf{x}^k$ is obtained when three processors finish their tasks. It can be seen that if the user needs the exact result $\mathbf{F}\mathbf{x}^k$, he must wait

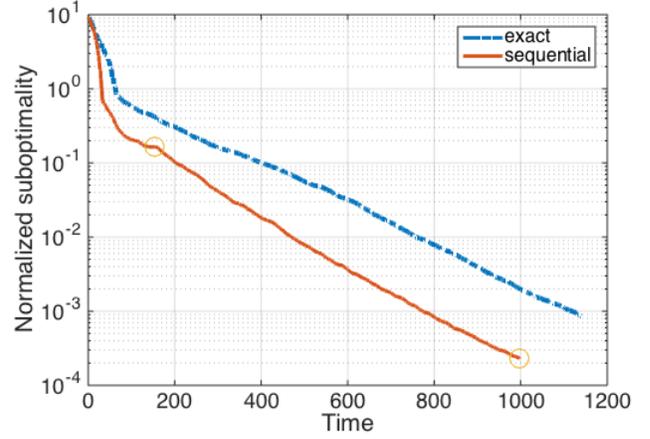


Fig. 3. Numerical results for Example 1, showing the normalized suboptimality $\|\mathbf{x}^k - \mathbf{x}^*\| / \|\mathbf{x}^*\|$ versus overall computation time. The solid line denotes the sequential approximation algorithm and the dashed line denotes the original proximal gradient method. A black circles denotes the end of one approximation level. To reach a normalized suboptimality 10^{-3} , the sequential approximation is roughly 30% faster than the original algorithm.

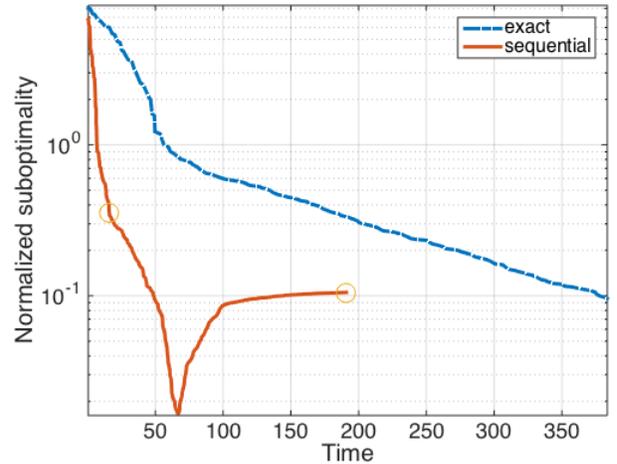


Fig. 4. Numerical results for Example 2, showing the normalized suboptimality $\|\mathbf{x}^k - \mathbf{x}^*\| / \|\mathbf{x}^*\|$ versus overall computation time. The solid line denotes the sequential approximation algorithm and the dashed line denotes the original proximal gradient method. In this example, the algorithm never uses the exact matrix \mathbf{F} , hence \mathbf{x}^k does not converge to the optimal solution of the original problem, but an approximate solution with a suboptimality 0.1. In this case, the sequential approximation algorithm is almost two times faster than the original algorithm to reach this suboptimality.

for all four processors to finish. It can be calculated that the average waiting time for three processors is approximately 1.08, while the average waiting time for four processors is 2.08. The performance of the algorithm is given in Figure 3.

Example 2 (approximate solution): This approach is also useful when we only aim to obtain an approximate solution of the problem. In this example we execute the sequential approximation algorithm with two levels of approximation where $\text{rank}(\mathbf{F}_{(1)}) = 5$ and $\text{rank}(\mathbf{F}_{(2)}) = 15$. This is implemented with a configuration $(k_1 = 5, k_2 = 10, k_3 = 0, k_4 = 0)$. In

other words, the user can recover $\mathbf{F}_{(1)}\mathbf{x}^k$ when any processor returns the result and can recover $\mathbf{F}_{(2)}\mathbf{x}^k$ when any 2 processors return the results. The average waiting time for one processor is approximately 0.25 and the average waiting time for two processors is approximately 0.58. Notice that in this case, the sequential approximation algorithm cannot converge to the true minimizer since it does not use the exact matrix \mathbf{F} in any level. Nevertheless the simulation results show that it gives a fairly good approximate solution. The performance of the algorithm is given in Figure 4.

V. APPENDIX

Proof of Theorem 1: It is easy to see that each matrix \mathbf{A}_i is encoded into at most $\lfloor k_i/i \rfloor + 1$ matrices $\tilde{\mathbf{A}}^{(1)}, \dots, \tilde{\mathbf{A}}^{(\lfloor k_i/i \rfloor + 1)}$, whose total number of rows sums up to s_i as defined in (6). Hence all the encoded data can be accommodated in the L matrices if it satisfies $\sum_{i=1}^L s_i \leq nL$.

Now we argue that this coding scheme possesses the sequential property in Property 1. Since the rows of $\mathbf{A}_i^{(j)}$, $j = 1, \dots, \lfloor k_i/i \rfloor$ are encoded with an (L, i) MDS code into the matrix $\tilde{\mathbf{A}}_i^{(j)}$, it is to see that any i entries of the matrix-vector multiplication $\tilde{\mathbf{A}}_i^{(j)}\mathbf{x}$ allow us to recover $\mathbf{A}_i^{(j)}\mathbf{x}$. Recall that each row of $\tilde{\mathbf{A}}_i^{(j)}$ is distributed to one processor, hence if i processors return the their matrix-multiplication results, the user can recover $\mathbf{A}_i^{(j)}\mathbf{x}$, and this holds for all $j = 1, \dots, \lfloor k_i/i \rfloor$.

If i does not divide k_i , we have the extra matrix $\tilde{\mathbf{A}}_i^{(\lfloor k_i/i \rfloor + 1)}$ whose rows are encoded using an $((L - i + \text{mod}(k_i, i)), \text{mod}(k_i, i))$ MDS codes into $\tilde{\mathbf{A}}_i^{(\lfloor k_i/i \rfloor + 1)}$ with $(L - i + \text{mod}(k_i, i))$ rows, and each row is distributed to one processor. It can be seen that any i processors will contain at least $\text{mod}(k_i, i)$ rows of $\tilde{\mathbf{A}}_i^{(\lfloor k_i/i \rfloor + 1)}$. Indeed, there are $i - \text{mod}(k_i, i)$ processors which do not store any row of $\tilde{\mathbf{A}}_i^{(\lfloor k_i/i \rfloor + 1)}$. The worst case is when a subset of i processors include all the processors which do not contain rows of $\tilde{\mathbf{A}}_i^{(\lfloor k_i/i \rfloor + 1)}$. Even in this case, we have the remaining $\text{mod}(k_i, i)$ processors which contain rows of $\tilde{\mathbf{A}}_i^{(\lfloor k_i/i \rfloor + 1)}$. This shows that it is always possible to recover $\mathbf{A}_i^{(j)}\mathbf{x}$, $j = 1, \dots, \lfloor k_i/i \rfloor$ and $\mathbf{A}_i^{(\lfloor k_i/i \rfloor + 1)}\mathbf{x}$ (hence $\mathbf{A}_i\mathbf{x}$) with any i processors. This argument holds for all $i, i = 1, \dots, L$ and thus concludes the proof. ■

Proof sketch of Theorem 2: We will only consider linear coding scheme in this paper where the matrix \mathbf{B}_i is a linear function of matrices $\mathbf{A}_1, \dots, \mathbf{A}_L$. It can be argued that, any linear coding scheme can be reduced to a scheme where each row of \mathbf{B}_i only consists of rows of one matrix \mathbf{A}_i (as shown in the Example in Section III-A). In other words, any linear scheme can be equivalently implemented as

$$\mathbf{B}_i = \mathbf{G}_i \begin{pmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_L \end{pmatrix} \quad (13)$$

where $\mathbf{G}_i \in \mathbb{R}^{n \times \sum_{j=1}^L k_j}$ is of the form

$$\mathbf{G}_i = \begin{pmatrix} \mathbf{G}_{i,1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_{i,2} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{G}_{i,L} \end{pmatrix}$$

for some $\mathbf{G}_{i,j} \in \mathbb{R}^{n_j \times k_i}$ where $\sum_{j=1}^L n_j = n$.

Since each row of \mathbf{B}_ℓ is used for encoding only one matrix \mathbf{A}_i , we use $n_{i,\ell}$ to denote the number of rows used for encoding \mathbf{A}_i in processor ℓ (which contains \mathbf{B}_ℓ). Let s_i be the number of rows used for the matrix \mathbf{A}_i across all L processors, i.e., $s_i = \sum_{\ell=1}^L n_{i,\ell}$. Now we show that in order to have a valid coding scheme for the distributed storage problem, s_i should satisfy

$$s_i \geq \begin{cases} \frac{k_i}{i} L & i \text{ divides } k_i \\ \lfloor \frac{k_i}{i} \rfloor \cdot L + L - i + \text{mod}(k_i, i) & \text{otherwise} \end{cases}$$

which matches the achievable coding scheme in (6).

First notice that in order to recover $\mathbf{A}_i\mathbf{z}$ using any i processors, a necessary condition on $n_{i,\ell}$ is

$$\sum_{\ell \in \mathcal{T}} n_{i,\ell} \geq k_i \quad (14)$$

for any subset $\mathcal{T} \subseteq [L]$ with $|\mathcal{T}| = i$, simply because $\mathbf{A}_i\mathbf{z}$ has length k_i . Hence a lower bound \hat{s}_i on s_i is given by the following optimization problem

$$\begin{aligned} & \text{minimize } \hat{s}_i := \sum_{\ell=1}^L n_{i,\ell} & (15) \\ & \text{s. t. } \sum_{\ell \in \mathcal{T}} n_{i,\ell} \geq k_i \text{ for all } \mathcal{T} \subseteq [L] \text{ with } |\mathcal{T}| = i \\ & n_{i,\ell} \in \mathbb{Z} \text{ for all } \ell \in [L] & (16) \end{aligned}$$

If we ignore the integer constraint on $n_{i,\ell}$, it is easy to see that the optimal solution to the relaxed linear programming problem is given by

$$n_{i,\ell}^* = k_i/i \text{ for all } \ell \in [L]$$

and the lower bound \hat{s}_i is equal to Lk_i/i . In the special case when i divides k_i , the optimal solution $n_{i,\ell}^*$ is an integer hence is also an optimal solution to the original problem (15). This shows that we have $s_i \geq \hat{s}_i = Lk_i/i$ if i divides k_i .

If i does not divide k_i , it can be argued that, due to the complete symmetry of the problem (15), the optimal solution of the integer programming problem (15) satisfies

$$n_{i,\ell}^* = \lceil k_i/i \rceil \text{ or } n_{i,\ell}^* = \lfloor k_i/i \rfloor \text{ for all } \ell \in [L]$$

Moreover, at most $\alpha := i - \text{mod}(k_i, i)$ among the L processors are allowed to dedicate $\lfloor k_i/i \rfloor$ rows to \mathbf{a}_i , and all other processors must dedicate $\lceil k_i/i \rceil$ rows to \mathbf{A}_i . Indeed, if we have $n_{i,\ell}^* = \lfloor k_i/i \rfloor$ for all $\ell \in \mathcal{T}' \subseteq [L]$ for some set \mathcal{T}' with

$|\mathcal{T}'| = \alpha + p$ for some $p \geq 1$, then for a set $\mathcal{T} \supseteq \mathcal{T}'$ we have

$$\begin{aligned} \sum_{\ell \in \mathcal{T}} n_{i,\ell} &= \sum_{\ell \in \mathcal{T}'} n_{i,\ell} + \sum_{\ell \in \mathcal{T} \setminus \mathcal{T}'} n_{i,\ell} \\ &= (\alpha + p) \lfloor k_i/i \rfloor + (i - (\alpha + p))(\lfloor k_i/i \rfloor + 1) \\ &= i \lfloor k_i/i \rfloor + \text{mod}(k_i, i) - p \\ &< i \lfloor k_i/i \rfloor + \text{mod}(k_i, i) = k_i, \end{aligned}$$

hence not be able to recover $\mathbf{A}_i \mathbf{z}$. We conclude that \hat{s}_i must satisfy

$$\begin{aligned} \hat{s}_i &= \sum_{\ell=1}^L n_{i,\ell} \geq \alpha \lfloor k_i/i \rfloor + (L - \alpha)(\lfloor k_i/i \rfloor + 1) \\ &= L \lfloor k_i/i \rfloor + L - i + \text{mod}(k_i, i) \end{aligned}$$

This shows a lower bound on s_i for the case when i does not divide k_i . ■

REFERENCES

- [1] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, p. 74, Feb. 2013.
- [2] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding Up Distributed Machine Learning Using Codes," *arXiv:1512.02673 [cs, math]*, Dec. 2015, arXiv: 1512.02673. [Online]. Available: <http://arxiv.org/abs/1512.02673>
- [3] N. S. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sep. 2016, pp. 954–960.
- [4] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication," *arXiv:1705.10464 [cs, math]*, May 2017, arXiv: 1705.10464. [Online]. Available: <http://arxiv.org/abs/1705.10464>
- [5] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sep. 2015, pp. 964–971.
- [6] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding," *arXiv:1612.03301 [cs, math, stat]*, Dec. 2016, arXiv: 1612.03301. [Online]. Available: <http://arxiv.org/abs/1612.03301>
- [7] 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 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 2100–2108.
- [8] C. Karakus, Y. Sun, and S. Diggavi, "Encoded distributed optimization," in *International Symposium on Information Theory (ISIT)*, 2017.
- [9] N. Parikh and S. Boyd, "Proximal Algorithms," *Found. Trends Optim.*, vol. 1, no. 3, pp. 127–239, Jan. 2014.
- [10] R. Puri and K. Ramchandran, "Multiple description source coding using forward error correction codes," in *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers*, vol. 1, Oct. 1999, pp. 342–346 vol.1.