Speeding Up Distributed Machine Learning Using Codes

Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran, *Fellow, IEEE*

***Abstract*— Codes are widely used in many engineering applications to offer *robustness* against *noise*. In large-scale systems, there are several types of noise that can affect the performance of distributed machine learning algorithms—straggler nodes, system failures, or communication bottlenecks—but there has been little interaction cutting across codes, machine learning, and distributed systems. In this paper, we provide theoretical insights on how *coded* solutions can achieve significant gains compared with uncoded ones. We focus on two of the most basic building blocks of distributed learning algorithms: *matrix multiplication* and *data shuffling*. For matrix multiplication, we use codes to alleviate the effect of stragglers and show that if the number of homogeneous workers is n, and the runtime of each subtask has an exponential tail, coded computation can speed up distributed matrix multiplication by a factor of $\log n$. For data shuffling, we use codes to reduce communication bottlenecks, exploiting the excess in storage. We show that when a constant fraction α of the data matrix can be cached at each worker, and n is the number of workers, *coded shuffling* reduces the communication cost by a factor of $(\alpha+\frac{1}{n})\gamma(n)$ compared with uncoded shuffling, where $\gamma(n)$ is the ratio of the cost of unicasting n messages to n users to multicasting a common message (of the same size) to n users. For instance, $\gamma(n) \simeq n$ if multicasting a message to n users is as cheap as unicasting a message to one user. We also provide experimental results, corroborating our theoretical gains of the coded algorithms.**

***Index Terms*— Algorithm design and analysis, channel coding, distributed computing, distributed databases, encoding, machine learning algorithms, multicast communication, robustness, runtime.**

Manuscript received October 18, 2016; revised May 24, 2017 and July 18, 2017; accepted July 19, 2017. Date of publication August 4, 2017; date of current version February 15, 2018. This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2017-0-00694, Coding for High-Speed Distributed Networks), the Brain Korea 21 Plus Project, and NSF CIF grant (No.1703678, Foundations of coding for modern distributed computing). This paper was presented in part at the 2015 Neural Information Processing Systems Workshop on Machine Learning Systems [1] and the 2016 IEEE International Symposium on Information Theory [2].

K. Lee is with the School of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea (e-mail: kw1jjang@kaist.ac.kr).

M. Lam and K. Ramchandran are with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: agnusmaximus@berkeley.edu; kannanr@eecs.berkeley.edu).

R. Pedarsani is with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA 93106 USA (e-mail: ramtin@ece.ucsb.edu).

D. Papailiopoulos is with the Department of Electrical and Computer Engineering, University of Wisconsin–Madison, Madison, WI 53706 USA (e-mail: dimitris@papail.io).

Communicated by P. Sadeghi, Associate Editor for Coding Techniques.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIT.2017.2736066

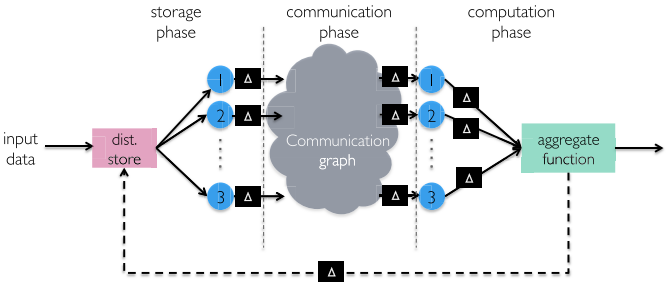


Fig. 1. **Conceptual diagram of the phases of distributed computation.** The algorithmic workflow of distributed (potentially iterative) tasks, can be seen as receiving input data, storing them in distributed nodes, communicating data around the distributed network, and then computing locally a function at each distributed node. The main bottlenecks in this execution (communication, stragglers, system failures) can all be abstracted away by incorporating a notion of delays between these phases, denoted by Δ boxes.

I. Introduction

In RECENT years, the computational paradigm for large-scale machine learning and data analytics has shifted towards massively large distributed systems, comprising individually small and unreliable computational nodes (low-end, commodity hardware). Specifically, modern distributed systems like Apache Spark [3] and computational primitives like MapReduce [4] have gained significant traction, as they enable the execution of production-scale tasks on data sizes of the order of petabytes. However, it is observed that the performance of a modern distributed system is significantly affected by anomalous system behavior and bottlenecks [5], i.e., a form of "system noise". Given the individually unpredictable nature of the nodes in these systems, we are faced with the challenge of securing fast and high-quality algorithmic results in the face of uncertainty.

In this work, we tackle this challenge using *coding theoretic* techniques. The role of codes in providing resiliency against noise has been studied for decades in several other engineering contexts, and is part of our everyday infrastructure (smartphones, laptops, WiFi and cellular systems, etc.). The goal of our work is to apply coding techniques to blueprint robust distributed systems, especially for distributed machine learning algorithms. The workflow of distributed machine learning algorithms in a large-scale system can be decomposed into three functional phases: a storage, a communication, and a computation phase, as shown in Fig. 1. In order to develop and deploy sophisticated solutions and tackle large-scale problems in machine learning, science, engineering, and commerce, it is important to understand and optimize novel and complex trade-offs across the multiple dimensions of

computation, communication, storage, and the accuracy of results. Recently, codes have begun to transform the storage layer of distributed systems in modern data centers under the umbrella of regenerating and locally repairable codes for distributed storage [6]–[21] which are also having a major impact on industry [22]–[25].

In this paper, we explore the use of coding theory to remove bottlenecks caused during the other phases: *the communication and computation phases* of distributed algorithms. More specifically, we identify two core blocks relevant to the communication and computation phases that we believe are key primitives in a plethora of distributed data processing and machine learning algorithms: *matrix multiplication* and *data shuffling*.

For matrix multiplication, we use codes to leverage the plethora of nodes and alleviate the effect of *stragglers*, i.e., nodes that are significantly slower than average. We show analytically that if there are n workers having identically distributed computing time statistics that are exponentially distributed, the optimal *coded matrix multiplication* is $\Theta(\log n)$[1] times faster than the uncoded matrix multiplication on average.

Data shuffling is a core element of many machine learning applications, and is well-known to improve the statistical performance of learning algorithms. We show that codes can be used in a novel way to trade off excess in available storage for reduced communication cost for data shuffling done in parallel machine learning algorithms. We show that when a constant fraction of the data matrix can be cached at each worker, and n is the number of workers, *coded shuffling* reduces the communication cost by a factor $\Theta(\gamma(n))$ compared to uncoded shuffling, where $\gamma(n)$ is the ratio of the cost of unicasting n messages to n users to multicasting a common message (of the same size) to n users. For instance, $\gamma(n) \simeq n$ if multicasting a message to n users is as cheap as unicasting a message to one user.

We would like to remark that a major innovation of our coding solutions is that they are woven into the fabric of the algorithmic design, and coding/decoding is performed over the representation field of the input data (e.g., floats or doubles). In sharp contrast to most coding applications, we do not need to "re-factor code" and modify the distributed system to accommodate for our solutions; it is all done seamlessly in the algorithmic design layer, an abstraction that we believe is much more impactful as it is located "higher up" in the system layer hierarchy compared to traditional applications of coding that need to interact with the stored and transmitted "bits" (e.g., as is the case for coding solutions for the physical or storage layer).

Overview of The Main Results

We now provide a brief overview of the main results of this paper. The following toy example illustrates the main idea of *Coded Computation*. Consider a system with three worker nodes and one master node, as depicted in Fig. 2. The goal is to compute a matrix multiplication $\mathbf{AX}$ for data matrix $\mathbf{A} \in \mathbb{R}^{q \times r}$ and input matrix $\mathbf{X} \in \mathbb{R}^{r \times s}$. The data matrix $\mathbf{A}$ is divided into two submatrices $\mathbf{A}_1 \in \mathbb{R}^{q/2 \times r}$ and $\mathbf{A}_2 \in \mathbb{R}^{q/2 \times r}$ and stored in node 1 and node 2, as shown in Fig. 2. The sum of the two submatrices is stored in node 3. After the master node transmits $\mathbf{X}$ to the worker nodes, each node computes the matrix multiplication of the stored matrix and the received matrix $\mathbf{X}$, and sends the computation result back to the master node. The master node can compute $\mathbf{AX}$ as soon as it receives *any* two computation results.

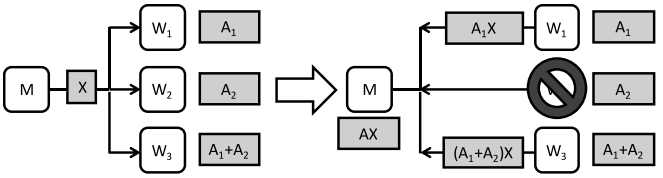


Fig. 2. **Illustration of *Coded Matrix Multiplication*.** Data matrix $\mathbf{A}$ is partitioned into 2 submatrices: $\mathbf{A}_1$ and $\mathbf{A}_2$. Node W_1 stores $\mathbf{A}_1$, node W_2 stores $\mathbf{A}_2$, and node W_3 stores $\mathbf{A}_1 + \mathbf{A}_2$. Upon receiving $\mathbf{X}$, each node multiplies $\mathbf{X}$ with the stored matrix, and sends the product to the master node. Observe that the master node can always recover $\mathbf{AX}$ upon receiving *any* 2 products, without needing to wait for the slowest response. For instance, consider a case where the master node has received $\mathbf{A}_1\mathbf{X}$ and $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$. By subtracting $\mathbf{A}_1\mathbf{X}$ from $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$, it can recover $\mathbf{A}_2\mathbf{X}$ and hence $\mathbf{AX}$.

Coded Computation designs parallel tasks for a linear operation using erasure codes such that its runtime is not affected by up to a certain number of stragglers. Matrix multiplication is one of the most basic linear operations and is the workhorse of a host of machine learning and data analytics algorithms, e.g., gradient descent based algorithm for regression problems, power-iteration like algorithms for spectral analysis and graph ranking applications, etc. Hence, we focus on the example of matrix multiplication in this paper. With coded computation, we will show that the runtime of the algorithm can be significantly reduced compared to that of other uncoded algorithms. The main result on *Coded Computation* is stated in the following (informal) theorem.

Theorem 1 (Coded Computation): If the number of workers is n, and the runtime of each subtask has an exponential tail, the optimal coded matrix multiplication is $\Theta(\log n)$ *times faster than the uncoded matrix multiplication.*

For the formal version of the theorem and its proof, see Sec. III-D.

We now overview the main results on coded shuffling. Consider a master-worker setup where a master node holds the entire data set. The generic machine learning task that we wish to optimize is the following: 1) the data set is randomly permuted and partitioned in batches at the master; 2) the master sends the batches to the workers; 3) each worker uses its batch and locally trains a model; 4) the local models are averaged at the master and the process is repeated. To reduce communication overheads between master and workers, *Coded Shuffling* exploits *i)* the locally cached data points of previous passes and *ii)* the "transmission strategy" of the master node.

We illustrate the basics of *Coded Shuffling* with a toy example. Consider a system with two worker nodes and one master node. Assume that the data set consists of 4 batches $\mathbf{A}_1, \ldots, \mathbf{A}_4$, which are stored across two workers as shown in Fig. 3. The sole objective of the master is to transmit $\mathbf{A}_3$ to the

[1]For any two sequences $f(n)$ and $g(n)$: $f(n) = \Omega(g(n))$ if there exists a positive constant c such that $f(n) \geq cg(n)$; $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

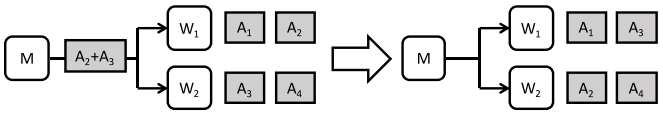


Fig. 3. **Illustration of *Coded Shuffling*.** Data matrix $\mathbf{A}$ is partitioned into 4 submatrices: $\mathbf{A}_1$ to $\mathbf{A}_4$. Before shuffling, worker W_1 has $\mathbf{A}_1$ and $\mathbf{A}_2$ and worker W_2 has $\mathbf{A}_3$ and $\mathbf{A}_4$. The master node can send $\mathbf{A}_2 + \mathbf{A}_3$ in order to shuffle the data stored at the two workers.

first worker and $\mathbf{A}_4$ to the second. For this purpose, the master node can simply multicast a *coded* message $\mathbf{A}_2 + \mathbf{A}_3$ to the worker nodes since the workers can decode the desired batches using the stored batches. Compared to the naïve (or uncoded) shuffling scheme in which the master node transmits $\mathbf{A}_2$ and $\mathbf{A}_3$ separately, this new shuffling scheme can save 50% of the communication cost, speeding up the overall machine learning algorithm. The *Coded Shuffling* algorithm is a generalization of the above toy example, which we explain in detail in Sec. IV.

Note that the above example assumes that *multicasting* a message to all workers costs exactly the same as unicasting a message to one of the workers. In general, we capture the advantage of using multicasting over unicasting by defining $\gamma(n)$ as follows:

$$\gamma(n) \stackrel{\text{def}}{=} \frac{\text{cost of unicasting } n \text{ separate msgs to } n \text{ workers}}{\text{cost of multicasting a common msg to } n \text{ workers}}. \tag{1}$$

Clearly, $1 \leq \gamma(n) \leq n$: if $\gamma(n) = n$, the cost of multicasting is equal to that of unicasting a single message (as in the above example); if $\gamma(n) = 1$, there is essentially no advantage of using multicast over unicast.

We now state the main result on *Coded Shuffling* in the following (informal) theorem.

Theorem 2 (Coded Shuffling): Let α be the fraction of the data matrix that can be cached at each worker, and n be the number of workers. Assume that the advantage of multicasting over unicasting is $\gamma(n)$. Then, coded shuffling reduces the communication cost by a factor of $\left(\alpha + \frac{1}{n}\right)\gamma(n)$ compared to uncoded shuffling.

For the formal version of the theorem and its proofs, see Sec. IV-D.

The remainder of this paper is organized as follows. In Sec. II, we provide an extensive review of the related works in the literature. Sec. III introduces the coded matrix multiplication, and Sec. IV introduces the coded shuffling algorithm. Finally, Sec. V presents conclusions and discusses open problems.

II. Related Work

A. Coded Computation and Straggler Mitigation

The straggler problem has been widely observed in distributed computing clusters. Dean and Barroso [5] show that running a computational task at a computing node often involves unpredictable latency due to several factors such as network latency, shared resources, maintenance activities, and power limits. Further, they argue that stragglers cannot be completely removed from a distributed computing cluster. Ananthanarayanan *et al.* [26] characterize the impact and causes of stragglers that arise due to resource contention, disk failures, varying network conditions, and imbalanced workload.

One approach to mitigate the adverse effect of stragglers is based on efficient straggler detection algorithms. For instance, the default scheduler of Hadoop constantly detects stragglers while running computational tasks. Whenever it detects a straggler, it relaunches the task that was running on the detected straggler at some other available node. Zaharia *et al.* [27] propose a modification to the existing straggler detection algorithm and show that the proposed solution can effectively reduce the completion time of MapReduce tasks. Ananthanarayanan *et al.* [26] propose a system that efficiently detects stragglers using real-time progress and cancels those stragglers, and show that the proposed system can further reduce the runtime of MapReduce tasks.

Another line of work is based on breaking the synchronization barriers in distributed algorithms [28], [29]. An asynchronous parallel execution can continuously make progress without having to wait for all the responses from the workers, and hence the overall runtime is less affected by stragglers. However, these asynchronous approaches break the serial consistency of the algorithm to be parallelized, and do not guarantee "correctness" of the end result, i.e., the output of the asynchronous algorithm can differ from that of a serial execution with an identical number of iterations.

Recently, replication-based approaches have been explored to tackle the straggler problem: by replicating tasks and scheduling the replicas, the runtime of distributed algorithms can be significantly improved [30]–[36]. By collecting outputs of the fast-responding nodes (and potentially canceling all the other slow-responding replicas), such replication-based scheduling algorithms can reduce latency. Lee *et al.* [35] show that even without replica cancellation, one can still reduce the average task latency by properly scheduling redundant requests. We view these policies as special instances of coded computation: such task replication schemes can be seen as *repetition-coded* computation. In Sec. III, we describe this connection in detail, and indicate that coded computation can significantly outperform replication (as is usually the case for coding vs. replication in other engineering applications).

Another line of work that is closely related to coded computation is about the latency analysis of coded distributed storage systems. Huang *et al.* [37] and Lee *et al.* [38] show that the flexibility of erasure-coded distributed storage systems allows for faster data retrieval performance than replication-based distributed storage systems. Joshi *et al.* [39] show that scheduling redundant requests to an increased number of storage nodes can improve the latency performance, and characterize the resulting storage-latency tradeoff. Sun *et al.* [40] study the problem of adaptive redundant requests scheduling, and characterize the optimal strategies for various scenarios. Kadhe [41] and Soljanin [42] analyze the latency performance of *availability codes*, a class of storage codes designed for enhanced availability. Joshi *et al.* [36] study the cost associated with scheduling of redundant requests, and propose a general scheduling policy that achieves a delicate balance between the latency performance and the cost.

We now review some recent works on coded computation, which have been published after our conference publications [1], [2]. In [43], an anytime coding scheme for approximate matrix multiplication is proposed, and it is shown that the proposed scheme can improve the quality of approximation compared with the other existing coded schemes for exact computation. Dutta *et al.* [44] propose a coded computation scheme called 'Short-Dot'. Short-Dot induces additional sparsity to the encoded matrices at the cost of reduced decoding flexibility, and hence potentially speeds up the computation. Tandon *et al.* [45] consider the problem of computing gradients in a distributed system, and propose a novel coded computation scheme tailored for computing a sum of functions. In many machine learning problems, the objective function is a sum of per-data loss functions, and hence the gradient of the objective function is the sum of gradients of per-data loss functions. Based on this observation, they propose *Gradient Coding*, which can reliably compute the exact gradient of any function in the presence of stragglers. While Gradient coding can be applied to computing gradients of any functions, it usually incurs significant storage and computation overheads. Bitar *et al.* [46] consider a secure coded computation problem where the input data matrices need to be secured from the workers. They propose a secure computation scheme based on Staircase codes, which can speed up the distributed computation while securing the input data from the workers. Lee *et al.* [47] consider the problem of large matrix-matrix multiplication, and propose a new coded computation scheme based on product codes. Reisizadehmobarakeh *et al.* [48] consider the coded computation problem on heterogenous computing clusters while our work assumes a homogeneous computing cluster. The authors show that by delicately distributing jobs across heterogenous workers, one can improve the performance of coded computation compared with the symmetric job allocation scheme, which is designed for homogeneous workers in our work. While most of the works focus on the application of coded computation to *linear* operations, a recent work shows that coding can be used also in distributed computing frameworks involving *nonlinear* operations [49]. Lee *et al.* [49] show that by leveraging the multi-core architecture in the worker computing units and "coding across" the multi-core computed outputs, significant (and in some settings unbounded) gains in speed-up in computational time can be achieved between the coded and uncoded schemes.

B. Data Shuffling and Communication Overheads

Distributed learning algorithms on large-scale networked systems have been extensively studied in the literature [50]–[60]. Many of the distributed algorithms that are implemented in practice share a similar algorithmic "anatomy": the data set is split among several cores or nodes, each node trains a model locally, then the local models are averaged, and the process is repeated. While training a model with parallel or distributed learning algorithms, it is common to randomly re-shuffle the data a number of times [29], [61]–[65]. This essentially means that after each shuffling the learning algorithm will go over the data in a different order than before. Although the effects of random shuffling are far from understood theoretically, the large statistical gains have turned it into a common practice. Intuitively, data shuffling before a new pass over the data, implies that nodes get a nearly "fresh" sample from the data set, which experimentally leads to better statistical performance. Moreover, bad orderings of the data—known to lead to slow convergence in the worst case [61], [64], [65]—are "averaged out". However, the statistical benefits of data shuffling do not come for free: each time a new shuffle is performed, the *entire* dataset is communicated over the network of nodes. This inevitably leads to performance bottlenecks due to heavy communication.

In this work, we propose to use coding opportunities to significantly reduce the communication cost of some distributed learning algorithms that require data shuffling. Our coded shuffling algorithm is built upon the coded caching scheme by Maddah-Ali and Niesen [66]. Coded caching is a technique to reduce the communication rate in content delivery networks. Mainly motivated by video sharing applications, coded caching exploits the multicasting opportunities between users that request different video files to significantly reduce the communication burden of the server node that has access to the files. Coded caching has been studied in many scenarios such as decentralized coded caching [67], online coded caching [68], hierarchical coded caching for wireless communication [69], and device-to-device coded caching [70]. Recently, Li *et al.* [71] proposed coded MapReduce that reduces the communication cost in the process of transferring the results of mappers to reducers.

Our proposed approach is significantly different from all related studies on coded caching in two ways: (i) we shuffle the *data points* among the computing nodes to *increase the statistical efficiency* of distributed computation and machine learning algorithms; and (ii) we *code the data over their actual representation* (i.e., over the doubles or floats) unlike the traditional coding schemes over bits. In Sec. IV, we describe how coded shuffling can remarkably speed up the communication phase of large-scale parallel machine learning algorithms, and provide extensive numerical experiments to validate our results.

The coded shuffling problem that we study is related to the index coding problem [72], [73]. Indeed, given a fixed "side information" reflecting the memory content of the nodes, the data delivery strategy for a particular permutation of the data rows induces an index coding problem. However, our coded shuffling framework is different from index coding in at least two significant ways. First, the coded shuffling framework involves multiple iterations of data being stored across all the nodes. Secondly, when the caches of the nodes are updated in coded shuffling, the system is unaware of the upcoming permutations. Thus, the cache update rules need to be designed to target any possible unknown permutation of data in succeeding iterations of the algorithm.

We now review some recent works on coded shuffling, which have been published after our first presentation [1], [2]. Attia and Tandon [74] study the information-theoretic limits of the coded shuffling problem. More specifically, the authors

completely characterize the fundamental limits for the case of 2 workers and the case of 3 workers. Attia and Tandon [75] consider the worse-case formulation of the coded shuffling problem, and propose a two-stage shuffling algorithm. Song and Fragouli [76] propose a new coded shuffling scheme based on pliable index coding. While most of the existing works focus on either coded computation or coded shuffling, one notable exception is [77]. In this work, the authors generalize the original coded MapReduce framework by introducing stragglers to the computation phases. Observing that highly flexible codes are not favorable to coded shuffling while replication codes allow for efficient shuffling, the authors propose an efficient way of coding to mitigate straggler effects as well as reduce the shuffling overheads.

III. Coded Computation

In this section, we propose a novel paradigm to mitigate the straggler problem. The core idea is simple: *we introduce redundancy into subtasks of a distributed algorithm such that the original task's result can be decoded from a subset of the subtask results, treating uncompleted subtasks as* ***erasures***. For this specific purpose, we use *erasure codes* to design *coded* subtasks.

An erasure code is a method of introducing redundancy to information for robustness to noise [78]. It encodes a message of k symbols into a longer message of n coded symbols such that the original k message symbols can be recovered by decoding a subset of coded symbols [78], [79]. We now show how erasure codes can be applied to distributed computation to mitigate the straggler problem.

A. Coded Computation

A coded distributed algorithm is specified by local functions, local data blocks, decodable sets of indices, and a decoding function: The local functions and data blocks specify the way the original computational task and the input data are distributed across n workers; and the decodable sets of indices and the decoding function are such that the desired computation result can be correctly recovered using the decoding function as long as the local computation results from any of the decodable sets are collected.

The formal definition of coded distributed algorithms is as follows.

Definition 1 (Coded Computation): Consider a computational task $f_{\mathbf{A}}(\cdot)$. A *coded* distributed algorithm for computing $f_{\mathbf{A}}(\cdot)$ is specified by

- local functions $\langle f^i_{\mathbf{A}_i}(\cdot)\rangle_{i=1}^n$ and local data blocks $\langle \mathbf{A}_i\rangle_{i=1}^n$;
- (minimal) decodable sets of indices $\mathcal{I} \subset \mathcal{P}([n])$ and a decoding function $\mathtt{dec}(\cdot,\cdot)$,

where $[n] \stackrel{\text{def}}{=} \{1, 2, \ldots, n\}$, and $\mathcal{P}(\cdot)$ is the power set of a set. The decodable sets of indices $\mathcal{I}$ is minimal: no element of $\mathcal{I}$ is a subset of other elements. The decoding function takes a sequence of indices and a sequence of subtask results, and it must correctly output $f_{\mathbf{A}}(\mathbf{x})$ if any decodable set of indices and its corresponding results are given.

A coded distributed algorithm can be run in a distributed computing cluster as follows. Assume that the i^{th} (encoded) data block $\mathbf{A}_i$ is stored at the i^{th} worker for all i. Upon receiving the input argument $\mathbf{x}$, the master node multicasts $\mathbf{x}$ to all the workers, and then waits until it receives the responses from any of the decodable sets. Each worker node starts computing its local function when it receives its local input argument, and sends the task result to the master node. Once the master node receives the results from some decodable set, it decodes the received task results and obtains $f_{\mathbf{A}}(\mathbf{x})$.

The algorithm described in Sec. I is an example of coded distributed algorithms: it is a coded distributed algorithm for matrix multiplication that uses an $(n, n-1)$ MDS code. One can generalize the described algorithm using an (n, k) MDS code as follows. For any $1 \le k \le n$, the data matrix $\mathbf{A}$ is first divided into k equal-sized submatrices.[2] Then, by applying an (n, k) MDS code to each element of the submatrices, n encoded submatrices are obtained. We denote these n encoded submatrices by $\mathbf{A}'_1, \mathbf{A}'_2, \ldots, \mathbf{A}'_n$. Note that the $\mathbf{A}'_i = \mathbf{A}_i$ for $1 \le i \le k$ if a systematic MDS code is used for the encoding procedure. Upon receiving *any* k task results, the master node can use the decoding algorithm to decode k task results. Then, one can find $\mathbf{AX}$ simply by concatenating them.

B. Runtime of Uncoded/Coded Distributed Algorithms

In this section, we analyze the runtime of uncoded and coded distributed algorithms. We first consider the overall runtime of an uncoded distributed algorithm, $T^{\text{uncoded}}_{\text{overall}}$. Assume that the runtime of each task is identically distributed and independent of others. We denote the runtime of the i^{th} worker under a computation scheme, say s, by T^{s}_i. Note that the distributions of T_i's can differ across different computation schemes.

$$T^{\text{uncoded}}_{\text{overall}} = T^{\text{uncoded}}_{(n)} \stackrel{\text{def}}{=} \max\{T^{\text{uncoded}}_1, \ldots, T^{\text{uncoded}}_n\}, \quad (2)$$

where $T_{(i)}$ is the i^{th} smallest one in $\{T_i\}_{i=1}^n$. From (2), it is clear that a single straggler can slow down the overall algorithm. A *coded* distributed algorithm is terminated whenever the master node receives results from any decodable set of workers. Thus, the overall runtime of a coded algorithm is *not* determined by the slowest worker, but by the first time to collect results from some decodable set in $\mathcal{I}$, i.e.,

$$T^{\text{coded}}_{\text{overall}} = T^{\text{coded}}_{(\mathcal{I})} \stackrel{\text{def}}{=} \min_{\mathbf{i}\in\mathcal{I}} \max_{j\in\mathbf{i}} T^{\text{coded}}_j \quad (3)$$

We remark that the runtime of uncoded distributed algorithms (2) is a special case of (3) with $\mathcal{I} = \{[n]\}$. In the following examples, we consider the runtime of the repetition-coded algorithms and the MDS-coded algorithms.

Example 1 (Repetition Codes): Consider an $\frac{n}{k}$-repetition-code where each local task is replicated $\frac{n}{k}$ times. We assume that each group of $\frac{n}{k}$ consecutive workers work on the replicas of one local task. Thus, the decodable sets of indices $\mathcal{I}$ are all the minimal sets that have k distinct task results, i.e., $\mathcal{I} = \{1, 2, \ldots, \frac{n}{k}\} \times \{\frac{n}{k}+1, \frac{n}{k}+2, \ldots, \frac{n}{k}+k\} \times \ldots \times \{n - \frac{n}{k}+1, n-\frac{n}{k}+2, \ldots, n\}$, where $A \times B$ denotes the Cartesian

[2] If the number of rows of $\mathbf{A}$ is not a multiple of k, one can append zero rows to $\mathbf{A}$ to make the number of rows a multiple of k.

product of matrix A and B. Thus,

$$T_{\text{overall}}^{\text{Repetition-coded}} = \max_{i\in[k]} \min_{j\in[\frac{n}{k}]} \{T_{(i-1)\frac{n}{k}+j}^{\text{Repetition-coded}}\}. \quad (4)$$

Example 2 (MDS Codes): If one uses an (n, k) MDS code, the decodable sets of indices are the sets of *any* k indices, i.e., $\mathcal{I} = \{\mathbf{i}|\mathbf{i} \subset [n],\ |\mathbf{i}| = k\}$. Thus,

$$T_{\text{overall}}^{\text{MDS-coded}} = T_{(k)}^{\text{MDS-coded}} \quad (5)$$

That is, the algorithm's runtime will be determined by the k^{th} response, not by the n^{th} response.

C. Probabilistic Model of Runtime

In this section, we analyze the runtime of uncoded/coded distributed algorithms assuming that task runtimes, including times to communicate inputs and outputs, are randomly distributed according to a certain distribution. For analytical purposes, we make a few assumptions as follows. We first assume the existence of the *mother runtime distribution* $F(t)$: we assume that running an algorithm using a *single* machine takes a random amount of time T_0, that is a positive-valued, continuous random variable parallelized according to F, i.e. $\Pr(T_0 \leq t) = F(t)$. We also assume that T_0 has a probability density function $f(t)$. Then, when the algorithm is distributed into a certain number of subtasks, say ℓ, the runtime distribution of each of the ℓ subtasks is assumed to be a scaled distribution of the mother distribution, i.e., $\Pr(T_i \leq t) = F(\ell t)$ for $1 \leq i \leq \ell$. Note that we are implicitly assuming a *symmetric* job allocation scheme, which is the optimal job allocation scheme if the underlying workers have the identical computing capabilities, i.e., homogeneous computing nodes are assumed. Finally, the computing times of the k tasks are assumed to be independent of one another.

Remark 1 (Homogeneous Clusters and Heterogenous Clusters): In this work, we assume homogeneous clusters: that is, all the workers have independent and identically distributed computing time statistics. While our symmetric job allocation is optimal for homogeneous cases, it can be strictly suboptimal for heterogenous cases. While our work focuses on homogeneous clusters, we refer the interested reader to a recent work [48] for a generalization of our problem setting to that of heterogeneous clusters, for which symmetric allocation strategies are no longer optimal.

We first consider an uncoded distributed algorithm with n (uncoded) subtasks. Due to the assumptions mentioned above, the runtime of each subtask is $F(nt)$. Thus, the runtime distribution of an uncoded distributed algorithm, denoted by $F_{\text{overall}}^{\text{uncoded}}(t)$, is simply $[F(nt)]^n$.

When repetition codes or MDS codes are used, an algorithm is first divided into k ($< n$) systematic subtasks, and then $n - k$ coded tasks are designed to provide an appropriate level of redundancy. Thus, the runtime of each task is distributed according to $F(kt)$. Using (4) and (5), one can easily find the runtime distribution of an $\frac{n}{k}$-repetition-coded distributed algorithm, $F_{\text{overall}}^{\text{Repetition}}$, and the runtime distribution of an (n, k)-MDS-coded distributed algorithm, $F_{\text{overall}}^{\text{MDS-coded}}$. For an $\frac{n}{k}$-repetition-coded distributed algorithm, one can first find the distribution of

$$\min_{j\in[\frac{n}{k}]} \{T_{(i-1)\frac{n}{k}+j}^{\text{Repetition-coded}}\},$$

and then find the distribution of the maximum of k such terms:

$$F_{\text{overall}}^{\text{Repetition}}(t) = \left[1 - [1 - F(kt)]^{\frac{n}{k}}\right]^k. \quad (6)$$

The runtime distribution of an (n, k)-MDS-coded distributed algorithm is simply the k^{th} order statistic:

$$F_{\text{overall}}^{\text{MDS-coded}}(t) = \int_{\tau=0}^{t} nkf(k\tau)\binom{n-1}{k-1} F(k\tau)^{k-1}\,[1 - F(k\tau)]^{n-k}\,d\tau. \quad (7)$$

Remark 2: For the same values of n and k, the runtime distribution of a repetition-coded distributed algorithm strictly dominates that of an MDS-coded distributed algorithm. This can be shown by observing that the decodable sets of the MDS-coded algorithm contain those of the repetition-coded algorithm.

In Fig. 4, we compare the runtime distributions of uncoded and coded distributed algorithms. We compare the runtime distributions of uncoded algorithm, repetition-coded algorithm, and MDS-coded algorithm with $n = 10$ and $k = 5$. In Fig. 4a, we use a shifted-exponential distribution as the mother runtime distribution. That is, $F(t) = 1 - e^{t-1}$ for $t \geq 1$. In Fig. 4b, we use the empirical task runtime distribution that is measured on an Amazon EC2 cluster.[3] Observe that for both cases, the runtime distribution of the MDS-coded distribution has the lightest tail.

D. Optimal Code Design for Coded Distributed Algorithms: The Shifted-Exponential Case

When a coded distributed algorithm is used, the original task is divided into a fewer number of tasks compared to the case of uncoded algorithms. Thus, the runtime of each task of a coded algorithm, which is $F(kt)$, is stochastically larger than that of an uncoded algorithm, which is $F(nt)$. If the value that we choose for k is too small, then the runtime of each task becomes so large that the overall runtime of the distributed coded algorithm will eventually increase. If k is too large, the level of redundancy may not be sufficient to prevent the algorithm from being delayed by the stragglers.

Given the mother runtime distribution and the code parameters, one can compute the overall runtime distribution of the coded distributed algorithm using (6) and (7). Then, one can optimize the design based on various target metrics, e.g., the expected overall runtime, the 99$^{\text{th}}$ percentile runtime, etc.

In this section, we show how one can design an optimal coded algorithm that minimizes *the expected overall runtime* for a shifted-exponential mother distribution. The shifted-exponential distribution strikes a good balance between accuracy and analytical tractability. This model is motivated by the model proposed in [80]: the authors used this distribution to model latency of file queries from cloud storage systems.

[3]The detailed description of the experiments is provided in Sec. III-F.

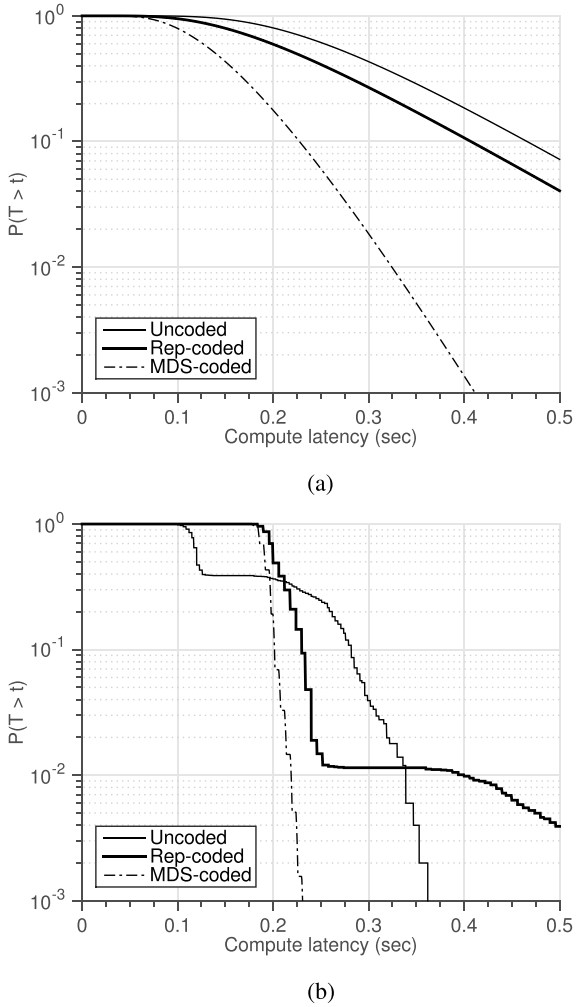


Fig. 4. **Runtime distributions of uncoded/coded distributed algorithms.** We plot the runtime distributions of uncoded/coded distributed algorithms. For the uncoded algorithms, we use $n = 10$, and for the coded algorithms, we use $n = 10$ and $k = 5$. In (a), we plot the runtime distribution when the runtime of tasks are distributed according to the shifted-exponential distribution. Indeed, the curves in (a) are analytically obtainable: See Sec. III-D for more details. In (b), we use the empirical task runtime distribution measured on an Amazon EC2 cluster.

The shifted-exponential distribution is the sum of a constant and an exponential random variable, i.e.,

$$\Pr(T_0 \leq t) = 1 - e^{-\mu(t-1)}, \quad \forall t \geq 1, \tag{8}$$

where the exponential rate μ is called the *straggling parameter*.

With this shifted-exponential model, we first characterize a lower bound on the fundamental limit of the average runtime.

Proposition 1: The average runtime of any distributed algorithm, in a distributed computing cluster with n workers, is lower bounded by $\frac{1}{n}$.

Proof: One can show that the average runtime of any distributed algorithm strictly decreases if the mother runtime distribution is replaced with a deterministic constant 1. Thus, the optimal average runtime with this deterministic mother distribution serves as a strict lower bound on the optimal average runtime with the shifted-exponential mother distribution. The constant mother distribution implies that stragglers do not exist, and hence the uncoded distributed algorithm achieves the optimal runtime, which is $\frac{1}{n}$. ■

We now analyze the average runtime of uncoded/coded distributed algorithms. We assume that n is large, and k is linear in n. Accordingly, we approximate $H_n \stackrel{\text{def}}{=} \sum_{i=1}^{n} \frac{1}{i} \simeq \log n$ and $H_{n-k} \simeq \log(n-k)$. We first note that the expected value of the maximum of n independent exponential random variables with rate μ is $\frac{H_n}{\mu}$. Thus, the average runtime of an uncoded distributed algorithm is

$$\mathbb{E}[T_{\text{overall}}^{\text{uncoded}}] = \frac{1}{n}\left(1 + \frac{1}{\mu}\log n\right) = \Theta\left(\frac{\log n}{n}\right). \tag{9}$$

For the average runtime of an $\frac{n}{k}$-Repetition-coded distributed algorithm, we first note that the minimum of $\frac{n}{k}$ independent exponential random variables with rate μ is distributed as an exponential random variable with rate $\frac{n}{k}\mu$. Thus,

$$\mathbb{E}[T_{\text{overall}}^{\text{Repetition-coded}}] = \frac{1}{k}\left(1 + \frac{k}{n\mu}\log k\right) = \Theta\left(\frac{\log n}{n}\right). \tag{10}$$

Finally, we note that the expected value of the k^{th} statistic of n independent exponential random variables of rate μ is $\frac{H_n - H_{n-k}}{\mu}$. Therefore,

$$\mathbb{E}[T_{\text{overall}}^{\text{MDS-coded}}] = \frac{1}{k}\left(1 + \frac{1}{\mu}\log\left(\frac{n}{n-k}\right)\right) = \Theta\left(\frac{1}{n}\right). \tag{11}$$

Using these closed-form expressions of the average runtime, one can easily find the optimal value of k that achieves the optimal average runtime. The following lemma characterizes the optimal repetition code for the repetition-coded algorithms and their runtime performances.

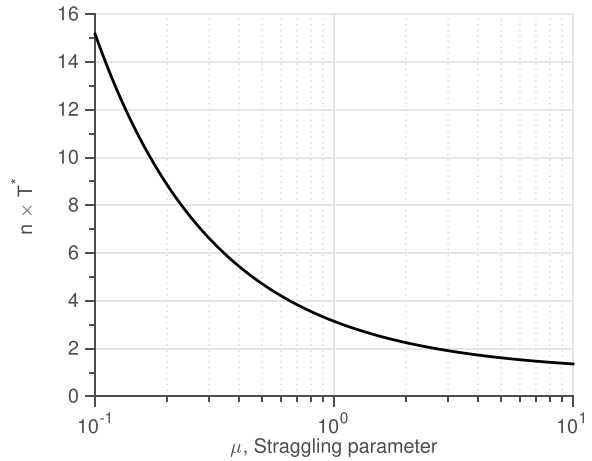
Lemma 1 (Optimal Repetition-Coded Distributed Algorithms): If $\mu \geq 1$, *the average runtime of an* $\frac{n}{k}$*-Repetition-coded distributed algorithm, in a distributed computing cluster with n workers, is minimized by setting* $k = n$, *i.e.,* not *replicating tasks. If* $\mu = \frac{1}{v}$ *for some integer* $v > 1$, *the average runtime is minimized by setting* $k = \mu n$, *and the corresponding minimum average runtime is* $\frac{1}{n\mu}(1 + \log(n\mu))$.

Proof: It is easy to see that (10) as a function of k has a unique extreme point. By differentiating (10) with respect to k and equating it to zero, we have $k = \mu n$. Thus, if $\mu \geq 1$, one should set $k = n$; if $\mu = \frac{1}{v} < 1$ for some integer v, one should set $k = \mu n$. ■

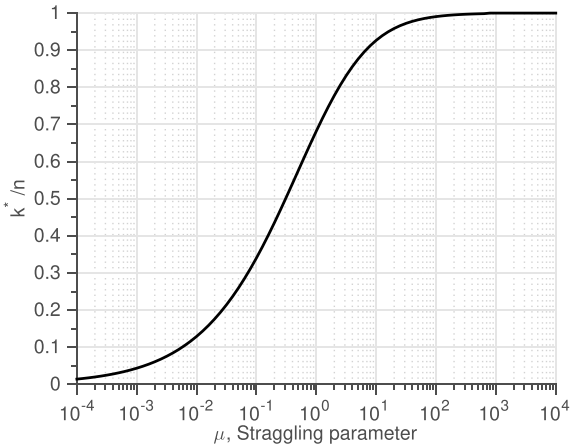
The above lemma reveals that the optimal repetition-coded distributed algorithm can achieve a lower average runtime than the uncoded distributed algorithm if $\mu < 1$; however, the optimal repetition-coded distributed algorithm still suffers from the factor of $\Theta(\log n)$, and cannot achieve the order-optimal performance. The following lemma, on the other hand, shows that the optimal MDS-coded distributed algorithm can achieve the order-optimal average runtime performance.

Lemma 2 (Optimal MDS-Coded Distributed Algorithms): The average runtime of an (n, k)*-MDS-coded distributed algorithm, in a distributed computing cluster with n workers, can be minimized by setting* $k = k^\star$ *where*

$$k^\star = \left\lceil 1 + \frac{1}{W_{-1}(-e^{-\mu-1})} \right\rceil n, \tag{12}$$



(a) $nT^\star$ as a function of μ.



(b) $\frac{k^\star}{n}$ as a function of μ.

Fig. 5. $nT^\star$ **and** $\frac{k^\star}{n}$ **as functions of** μ**.** As a function of the straggling parameter, we plot the normalized optimal computing time and the optimal value of k. (a) $nT^\star$ as a function of μ. (b) $\frac{k^\star}{n}$ as a function of μ.

and $W_{-1}(\cdot)$ *is the lower branch of Lambert W function*[4] *Thus,*

$$T^\star \stackrel{\text{def}}{=} \min_k \ \mathbb{E}[T^{\textit{MDS-coded}}_{\textit{overall}}] = \frac{-W_{-1}(-e^{-\mu-1})}{\mu n} \stackrel{\text{def}}{=} \frac{\gamma^\star(\mu)}{n}. \tag{13}$$

Proof: It is easy to see that (11) as a function of k has a unique extreme point. By differentiating (11) with respect to k and equating it to zero, we have $\frac{1}{k^\star}\left(1+\frac{1}{\mu}\log\left(\frac{n}{n-k^\star}\right)\right) = \frac{1}{\mu}\frac{1}{n-k^\star}$. By setting $k = \alpha^\star n$, we have $\frac{1}{\alpha^\star}\left(1+\frac{1}{\mu}\log\left(\frac{1}{1-\alpha^\star}\right)\right) = \frac{1}{\mu}\frac{1}{1-\alpha^\star}$, which implies $\mu+1 = \frac{1}{1-\alpha^\star} - \log\left(\frac{1}{1-\alpha^\star}\right)$. By defining $\beta = \frac{1}{1-\alpha^\star}$ and exponentiating both the sides, we have $e^{\mu+1} = \frac{e^\beta}{\beta}$. Note that the solution of $\frac{e^x}{x} = t$, $t \geq e$ and $x \geq 1$ is $x = -W_{-1}(-\frac{1}{t})$. Thus, $\beta = -W_{-1}(-e^{-\mu-1})$. By plugging the above equation into the definition of β, the claim is proved. ∎

We plot $nT^\star$ and $\frac{k^\star}{\mu}$ as functions of μ in Fig. 5.

[4] $W_{-1}(x)$, the lower branch of Lambert W function evaluated at x, is the unique solution of $te^t = x$ and $t \leq -1$.

In addition to the order-optimality of MDS-coded distributed algorithms, the above lemma precisely characterizes the gap between the achievable runtime and the optimistic lower bound of $\frac{1}{n}$. For instance, when $\mu > 1$, the optimal average runtime is only 3.15 away from the lower bound.

Remark 3 (Storage Overhead): So far, we have considered only the runtime performance of distributed algorithms. Another important metric to be considered is the storage cost. When coded computation is being used, the storage overhead may increase. For instance, the MDS-coded distributed algorithm for matrix multiplication, described in Sec. III-A, requires $\frac{1}{k}$ of the whole data to be stored at each worker, while the uncoded distributed algorithm requires $\frac{1}{n}$. Thus, the storage overhead factor is $\frac{\frac{1}{k}-\frac{1}{n}}{\frac{1}{n}} = \frac{n}{k} - 1$. If one uses the runtime-optimal MDS-coded distributed algorithm for matrix multiplication, the storage overhead is $\frac{n}{k^\star} - 1 = \frac{1}{\alpha^\star} - 1$.

E. Coded Gradient Descent: An MDS-Coded Distributed Algorithm for Linear Regression

In this section, as a concrete application of coded matrix multiplication, we propose the *coded gradient descent* for solving large-scale linear regression problems.

We first describe the (uncoded) gradient-based distributed algorithm. Consider the following linear regression,

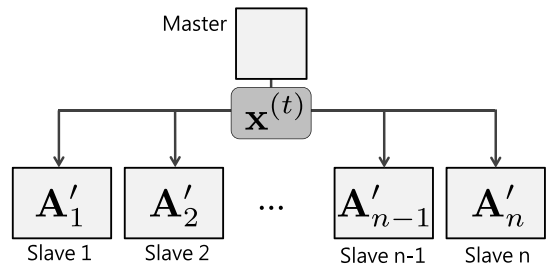
$$\min_{\mathbf{x}} f(\mathbf{x}) \stackrel{\text{def}}{=} \min_{\mathbf{x}} \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{y}\|_2^2, \tag{14}$$

where $\mathbf{y} \in \mathbb{R}^q$ is the label vector, $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_q]^T \in \mathbb{R}^{q\times r}$ is the data matrix, and $\mathbf{x} \in \mathbb{R}^r$ is the unknown weight vector to be found. We seek a distributed algorithm to solve this regression problem. Since $f(\mathbf{x})$ is convex in $\mathbf{x}$, the gradient-based distributed algorithm works as follows. We first compute the objective function's gradient: $\nabla f(\mathbf{x}) = \mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{y})$. Denoting by $\mathbf{x}^{(t)}$ the estimate of $\mathbf{x}$ after the t^{th} iteration, we iteratively update $\mathbf{x}^{(t)}$ according to the following equation.

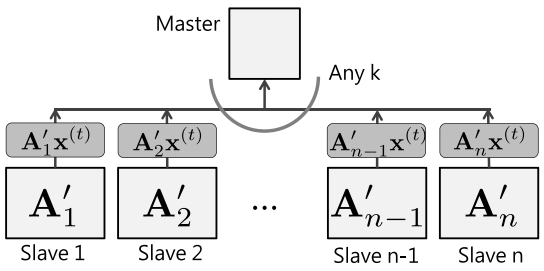
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta\nabla f(\mathbf{x}^{(t)}) = \mathbf{x}^{(t)} - \eta\mathbf{A}^T(\mathbf{A}\mathbf{x}^{(t)} - \mathbf{y}) \tag{15}$$

The above algorithm is guaranteed to converge to the optimal solution if we use a small enough step size η [81], and can be easily distributed. We describe one simple way of parallelizing the algorithm, which is implemented in many open-source machine learning libraries including Spark `mllib` [82]. As $\mathbf{A}^T(\mathbf{A}\mathbf{x}^{(t)} - \mathbf{y}) = \sum_{i=1}^q \mathbf{a}_i(\mathbf{a}_i^T\mathbf{x}^{(t)} - \mathbf{y}_i)$, gradients can be computed in a distributed way by computing partial sums at different worker nodes and then adding all the partial sums at the master node. This distributed algorithm is an *uncoded* distributed algorithm: in each round, the master node needs to wait for all the task results in order to compute the gradient.[5]

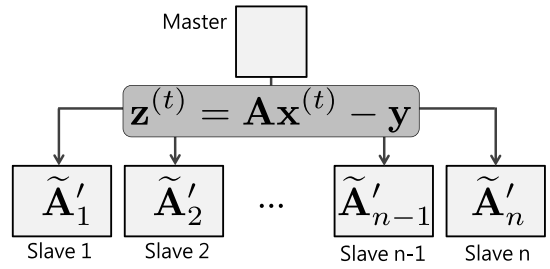
[5] Indeed, one may apply another coded computation scheme called Gradient Coding [45], which was proposed after our conference publications. By applying Gradient Coding to this algorithm, one can achieve straggler tolerance but at the cost of significant computation and storage overheads. More precisely, it incurs $\Theta(n)$ larger computation and storage overheads in order to protect the algorithm from $\Theta(n)$ stragglers. Later in this section, we will show that our coded computation scheme, which is tailor-designed for linear regression, incurs $\Theta(1)$ overheads to protect the algorithm from $\Theta(n)$ stragglers.



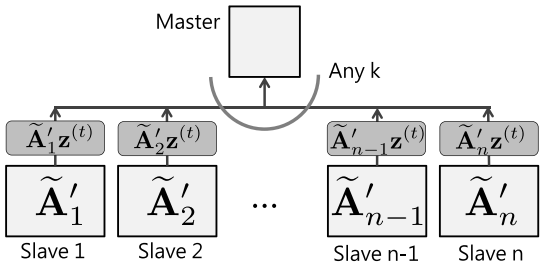
(a) In the beginning of the t^{th} iteration, the master node multicasts $\mathbf{x}^{(t)}$ to the worker nodes.



(b) The master node waits for the earliest responding k worker nodes, and computes $\mathbf{A}\mathbf{x}^{(t)}$.



(c) The master node computes $\mathbf{z}^{(t)}$ and multicasts it to the worker nodes.



(d) The master node waits for the k earliest responding worker nodes, and computes $\mathbf{A}^T\mathbf{z}^{(t)}$ or $\nabla f(\mathbf{x}^{(t)})$.

Fig. 6. **Illustration of a coded gradient descent approach for linear regression.** The coded gradient descent computes a gradient of the objective function using *coded matrix multiplication* twice: in each iteration, it first computes $\mathbf{A}\mathbf{x}^{(t)}$ as depicted in (a) and (b), and then computes $\mathbf{A}^T(\mathbf{A}\mathbf{x}^{(t)} - \mathbf{y})$ as depicted in (c) and (d).

Thus, the runtime of each update iteration is determined by the slowest response among all the worker nodes.

We now propose the *coded gradient descent*, a coded distributed algorithm for linear regression problems. Note that in each iteration, the following two matrix-vector multiplications are computed.

$$\mathbf{A}\mathbf{x}^{(t)}, \quad \mathbf{A}^T(\mathbf{A}\mathbf{x}^{(t)} - \mathbf{y}) \stackrel{\text{def}}{=} \mathbf{A}^T\mathbf{z}^{(t)} \tag{16}$$

In Sec. III-A, we proposed the MDS-coded distributed algorithm for matrix multiplication. Here, we apply the algorithm twice to compute these two multiplications in each iteration. More specifically, for the first matrix multiplication, we choose $1 \leq k_1 < n$ and use an (n, k_1)-MDS-coded distributed algorithm for matrix multiplication to encode the data matrix $\mathbf{A}$. Similarly for the second matrix multiplication, we choose $1 \leq k_2 < n$ and use a (n, k_2)-MDS-coded distributed algorithm to encode the transpose of the data matrix. Denoting the i^{th} row-split (column-split) of $\mathbf{A}$ as $\mathbf{A}_i$ ($\widetilde{\mathbf{A}}_i$), the i^{th} worker stores both $\mathbf{A}_i$ and $\widetilde{\mathbf{A}}_i$. In the beginning of each iteration, the master node multicasts $\mathbf{x}^{(t)}$ to the worker nodes, each of which computes the local matrix multiplication for $\mathbf{A}\mathbf{x}^{(t)}$ and sends the result to the master node. Upon receiving *any* k_1 task results, the master node can start decoding the result and obtain $\mathbf{z}^{(t)} = \mathbf{A}\mathbf{x}^{(t)}$. The master node now multicasts $\mathbf{z}^{(t)}$ to the workers, and the workers compute local matrix multiplication for $\mathbf{A}^T\mathbf{z}^{(t)}$. Finally, the master node can decode $\mathbf{A}^T\mathbf{z}^{(t)}$ as soon as it receives any k_2 task results, and can proceed to the next iteration. Fig. 6 illustrates the protocol with $k_1 = k_2 = n - 1$.

Remark 4 (Storage Overhead of the Coded Gradient Descent): The coded gradient descent requires each node to store a $(\frac{1}{k_1} + \frac{1}{k_2} - \frac{1}{k_1 k_2})$-fraction of the data matrix. As the minimum storage overhead per node is a $\frac{1}{n}$-fraction of the data matrix, the relative storage overhead of the coded gradient descent algorithm is at least about factor of 2, if $k_1 \simeq n$ and $k_2 \simeq n$.

F. Experimental Results

In order to see the efficacy of coded computation, we implement the proposed algorithms and test them on an Amazon EC2 cluster. We first obtain the empirical distribution of task runtime in order to observe how frequently stragglers appear in our testbed by measuring round-trip times between the master node and each of 10 worker instances on an Amazon EC2 cluster. Each worker computes a matrix-vector multiplication and passes the computation result to the master node, and the master node measures round trip times that include both computation time and communication time. Each worker repeats this procedure 500 times, and we obtain the empirical distribution of round trip times across all the worker nodes.

In Fig. 7, we plot the histogram and complementary CDF (CCDF) of measured computing times; the average round trip time is 0.11 second, and the 95$^{\text{th}}$ percentile latency is 0.20 second, i.e., roughly five out of hundred tasks are going to be roughly two times slower than the average tasks. Assuming the probability of a worker being a straggler is 5%, if one runs an uncoded distributed algorithm with 10 workers, the probability of not seeing such a straggler is only about 60%, so the algorithm is slowed down by a factor of more than 2 with probability 40%. Thus, this observation strongly emphasizes the necessity of an efficient straggler mitigation algorithm. In Fig. 4a, we plot the runtime distributions of uncoded/coded distributed algorithms using this empirical distribution as the

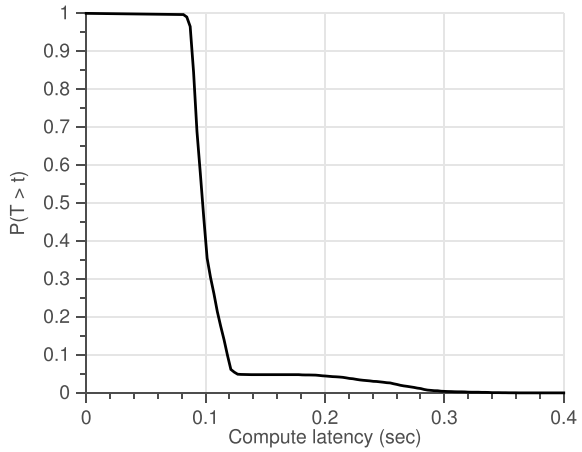


Fig. 7. **Empirical CCDF of the measured round trip times.** We measure round trip times between the master node and each of 10 worker nodes on an Amazon EC2 cluster. A round trip time consists of transmission time of the input vector from the master to a worker, computation time, and transmission time of the output vector from a worker to the master.

mother runtime distribution. When an uncoded distributed algorithm is used, the overall runtime distribution entails a heavy tail, while the runtime distribution of the MDS-coded algorithm has almost no tail.

We then implement the coded matrix multiplication in C++ using OpenMPI [83] and benchmark on a cluster of 26 EC2 instances (25 workers and a master).[6] Also, three uncoded matrix multiplication algorithms – block, column-partition, and row-partition – are implemented and benchmarked.

We randomly draw a square matrix of size 5750×5750, a fat matrix of size 5750×11500, and a tall matrix of size 11500×5750, and multiply them with a column vector. For the coded matrix multiplication, we choose an $(25, 23)$ MDS code so that the runtime of the algorithm is not affected by any 2 stragglers. Fig. 8 shows that the coded matrix multiplication outperforms all the other parallel matrix multiplication algorithms in most cases. On a cluster of `m1-small`, the most unreliable instances, the coded matrix multiplication achieves about 40% average runtime reduction and about 60% tail reduction compared to the best of the 3 uncoded matrix multiplication algorithmss. On a cluster of `c1-medium` instances, the coded algorithm achieves the best performance in most of the tested cases: the average runtime is reduced by at most 39.5%, and the 95$^{\text{th}}$ percentile runtime is reduced by at most 58.3%. Among the tested cases, we observe one case in which both the uncoded row-partition and the coded row-partition algorithms are outperformed by the uncoded column-partition algorithm. This is the case of a fat matrix multiplication with `c1-medium` instances. Note that when a row-partition algorithm is used, the size of messages from the master node to the workers is n times larger compared with the case of column-partition algorithms. Thus, when the variability of computational times becomes low compared with that of communication time, the larger communication overhead of row-partition algorithms seems to arise, nullifying the benefits of coding.

We also evaluate the performance of the coded gradient descent algorithm for linear regression. The coded linear regression procedure is also implemented in C++ using OpenMPI, and benchmarked on a cluster of 11 EC2 machines (10 workers and a master). Similar to the previous benchmarks, we randomly draw a square matrix of size 2000×2000, a fat matrix of size 400×10000, and a tall matrix of size 10000×400, and use them as a data matrix. We use a $(10, 8)$-MDS code for the coded linear regression so that each multiplication of the gradient descent algorithm is not slowed down by up to 2 stragglers. Fig. 9 shows that the gradient algorithm with the *coded matrix multiplication* significantly outperforms the one with the uncoded matrix multiplication; the average runtime is reduced by 31.3% to 35.7%, and the tail runtime is reduced by 27.9% to 35.6%.

[6]For the benchmark, we manage the cluster using the StarCluster toolkit [84]. Input data is generated using a Python script, and the input matrix is row-partitioned for each of the workers (with the required encoding as described in the previous sections) in a preprocessing step. The procedure begins by having all of the worker nodes read in their respective row-partitioned matrices. Then, the master node reads the input vector and distributes it to all worker nodes in the cluster through an asynchronous send (`MPI_Isend`). Upon receiving the input vector, each worker node begins matrix multiplication through a BLAS [85] routine call and once completed sends the result back to the master using `MPI_Send`. The master node waits for a sufficient number of results to be received by continuously polling (`MPI_Test`) to see if any results are obtained. The procedure ends when the master node decodes the overall result after receiving enough partial results.

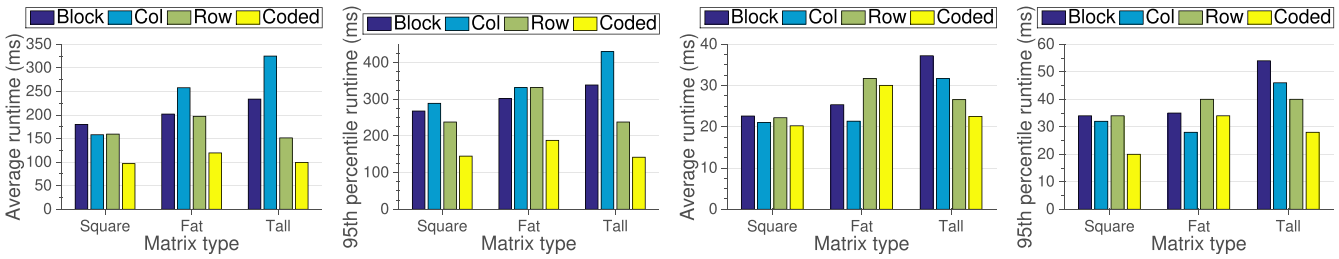
IV. Coded Shuffling

We shift our focus from solving the straggler problem to solving the communication bottleneck problem. In this section, we explain the problem of data-shuffling, propose the *Coded Shuffling* algorithm, and analyze its performance.

A. Setup and Notations

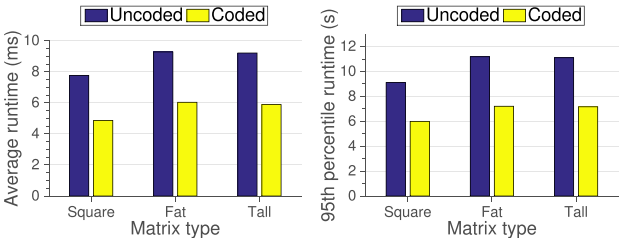
We consider a master-worker distributed setup, where the master node has access to the entire data-set. Before every *iteration* of the distributed algorithm, the master node randomly partition the entire data set into n subsets, say $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$. The goal of the shuffling phase is to distribute each of these partitioned data sets to the corresponding worker so that each worker can perform its distributed task with its own exclusive data set after the shuffling phase.

We let $\mathbf{A}(\mathcal{J}) \in \mathbb{R}^{|\mathcal{J}| \times r}$, $\mathcal{J} \subset [q]$ be the concatenation of $|\mathcal{J}|$ rows of matrix $\mathbf{A}$ with indices in $\mathcal{J}$. Assume that each worker node has a cache of size s data rows (or $s \times r$ real numbers). In order to be able to fully store the data matrix across the worker nodes, we impose the inequality condition $q/n \leq s$. Further, clearly if $s > q$, the data matrix can be fully stored at each worker node, eliminating the need for any shuffling. Thus, without loss of generality we assume that $s \leq q$. As explained earlier working on the same data points at each worker node in all the iterations of the iterative optimization algorithm leads to slow convergence. Thus, to enhance the statistical efficiency of the algorithm, the data matrix is shuffled after each iteration. More precisely, at each iteration t, the set of data rows $[q]$ is partitioned uniformly at random into n subsets S_i^t, $1 \leq i \leq n$ so that $\cup_{i=1}^n S_i^t = [q]$ and $S_i^t \cap S_j^t = \emptyset$ when $i \neq j$; thus, each worker node computes a fresh local function of the data. Clearly, the data set that



(a) `m1-small`, average runtime (b) `m1-small`, tail runtime (c) `c1-med`, average runtime (d) `c1-med`, tail runtime

Fig. 8. **Comparison of parallel matrix multiplication algorithms.** We compare various parallel matrix multiplication algorithms: block, column-partition, row-partition, and coded (row-partition) matrix multiplication. We implement the four algorithms using OpenMPI and test them on Amazon EC2 cluster of 25 instances. We measure the average and the 95$^{\text{th}}$ percentile runtime of the algorithms. Plotted in (a) and (b) are the results with `m1-small` instances, and in (c) and (d) are the results with `c1-medium` instances.



(a) (b)

Fig. 9. **Comparison of parallel gradient algorithms.** We compare parallel gradient algorithms for linear regression problems. We implement both the uncoded gradient descent algorithm and the coded gradient descent algorithm using Open MPI, and test them on an Amazon EC2 cluster of 10 worker instances. Plotted are the average and the 95$^{\text{th}}$ percentile runtimes of the algorithms. (a) Average runtime. (b) Tail runtime.

worker i works on has cardinality q/n, i.e., $|S_i^t| = q/n$. Note that the sampling we consider here is *without replacement*, and hence these data sets are non-overlapping.

B. Shuffling Schemes

We now present our coded shuffling algorithm, consisting of a transmission strategy for the master node, and caching and decoding strategies for the worker nodes. Let C_i^t be the cache content of node i (set of row indices stored in cache i) at the end of iteration t. We design a transmission algorithm (by the master node) and a cache update algorithm to ensure that (i) $S_i^t \subset C_i^t$; and (ii) $C_i^t \setminus S_i^t$ is distributed uniformly at random without replacement in the set $[q] \setminus S_i^t$. The first condition ensures that at each iteration, the workers have access to the data set that they are supposed to work on. The second condition provides the opportunity of effective coded transmissions for shuffling in the next iteration as will be explained later.

1) Cache Update Rule: We consider the following cache update rule: the new cache will contain the subset of the data points used in the current iteration (this is needed for the local computations), plus a random subset of the previous cached contents. More specifically, q/n rows of the new cache are precisely the rows in S_i^{t+1}, and $s - q/n$ rows of the cache are sampled points from the set $C_i^t \setminus S_i^{t+1}$, uniformly at random without replacement. Since the permutation π^t is picked uniformly at random, the marginal distribution of the cache contents at iteration $t+1$ given S_i^{t+1}, $1 \le i \le n$ is described as follows: $S_i^{t+1} \subset C_i^{t+1}$ and $C_i^{t+1} \setminus S_i^{t+1}$ is distributed uniformly at random in $[q] \setminus S_i^{t+1}$ without replacement.

2) Encoding and Transmission Schemes: We now formally describe two transmission schemes of the master node: (1) uncoded transmission and (2) coded transmission. In the following descriptions, we drop the iteration index t (and $t+1$) for the ease of notation.

The uncoded transmission first finds how many data rows in S_i are already cached in C_i, i.e. $|C_i \cap S_i|$. Since, the new permutation (partitioning) is picked uniformly at random, s/q fraction of the data row indices in S_i are cached in C_i, so as q gets large, we have $|C_i \cap S_i| + o(q) = \frac{q}{n}(1 - s/q)$. Thus, without coding, the master node needs to transmit $\frac{q}{n}(1-s/q)$ data points to each of the n worker nodes. The total communication rate (in data points transmitted per iteration) of the uncoded scheme is then

$$R_u = n \times \frac{q}{n}(1 - s/q) = q(1 - s/q). \tag{17}$$

We now describe the coded transmission scheme. Define the set of "exclusive" cache content as $\widetilde{C}_{\mathcal{I}} = (\cap_{i \in \mathcal{I}} C_i) \cap \left(\cap_{i' \in [n] \setminus \mathcal{I}} C_{i'}^{\mathsf{C}}\right)$ that denotes the set of rows that are stored at the caches of $\mathcal{I}$, and are *not* stored at the caches of $[n] \setminus \mathcal{I}$. For each subset $\mathcal{I}$ with $|\mathcal{I}| \ge 2$, the master node will multicast $\sum_{i \in \mathcal{I}} \mathbf{A}(S_i \cap \widetilde{C}_{\mathcal{I} \setminus \{i\}})$ to the worker nodes. Note that in general, the matrices $\mathbf{A}$'s differ in their sizes, so one has to zero-pad the shorter matrices and sum the zero-padded matrices. Algorithm 1 provides the pseudocode of the coded encoding and transmission scheme.[7]

3) Decoding Algorithm: The decoding algorithm for the uncoded transmission scheme is straightforward: each worker simply takes the additional data rows that are required for the new iteration, and ignores the other data rows. We now describe the decoding algorithm for the coded transmission scheme. Each worker, say worker i, decodes each encoded data row as follows. Consider an encoded data row for some $\mathcal{I}$ that contains i. (All other data rows are discarded.) Such an encoded data row must be the sum of some data row in S_i and $|\mathcal{I}| - 1$ data rows in $\widetilde{C}_{\mathcal{I} \setminus \{i\}}$, which are available in worker i by the definition of $\widetilde{C}$. Hence, the worker can always subtract

[7] Note that for each encoded data row, the master node also needs to transmit tiny metadata describing which data rows are included in the summation. We omit this detail in the description of the algorithm.

Algorithm 1 Coded Encoding and Transmission Scheme

procedure ENCODING($[C_i]_{i=1}^n$)
  for each $\mathcal{I} \in [n]^n$, $|\mathcal{I}| > 2$ **do**
    $\widetilde{C}_{\mathcal{I}} = (\cap_{i\in\mathcal{I}} C_i) \cap \left(\cap_{i'\in[n]\setminus\mathcal{I}} C_{i'}^{\complement}\right)$
    $\ell \leftarrow \max_{i=1}^{|\mathcal{I}|} |S_i \cap \widetilde{C}_{\mathcal{I}\setminus\{i\}}|$
    for each $i \in \mathcal{I}$ **do**
      $\mathbf{B}_i[1 : |S_i \cap \widetilde{C}_{\mathcal{I}\setminus\{i\}}|, :] \leftarrow \mathbf{A}(S_i \cap \widetilde{C}_{\mathcal{I}\setminus\{i\}})$
      $\mathbf{B}_i[|S_i \cap \widetilde{C}_{\mathcal{I}\setminus\{i\}}| + 1 : \ell, :] \leftarrow \mathbf{0}$
    end for
    broadcast $\sum_{i\in\mathcal{I}} \mathbf{B}_i$
  end for
end procedure

the data rows corresponding to $\widetilde{C}_{\mathcal{I}\setminus\{i\}}$ and decode the data row in S_i.

C. Example

The following example illustrates the coded shuffling scheme.

Example 3: Let $n = 3$. Recall that worker node i needs to obtain $\mathbf{A}(S_i \cap C_i^{\complement})$ for the next iteration of the algorithm. Consider $i = 1$. The data rows in $S_1 \cap C_1^{\complement}$ are stored either exclusively in C_2 or C_3 (i.e. $\widetilde{C}_2$ or $\widetilde{C}_3$), or stored in both C_2 and C_3 (i.e. $\widetilde{C}_{2,3}$). The transmitted message consists of 4 parts:

- (Part 1) $M_{\{1,2\}} = \mathbf{A}(S_1 \cap \widetilde{C}_2) + \mathbf{A}(S_2 \cap \widetilde{C}_1)$,
- (Part 2) $M_{\{1,3\}} = \mathbf{A}(S_1 \cap \widetilde{C}_3) + \mathbf{A}(S_3 \cap \widetilde{C}_1)$,
- (Part 3) $M_{\{2,3\}} = \mathbf{A}(S_2 \cap \widetilde{C}_3) + \mathbf{A}(S_3 \cap \widetilde{C}_2)$, and
- (Part 4) $M_{\{1,2,3\}} = \mathbf{A}(S_1 \cap \widetilde{C}_{2,3}) + \mathbf{A}(S_2 \cap \widetilde{C}_{1,3}) + \mathbf{A}(S_3 \cap \widetilde{C}_{1,2})$.

We show that worker node 1 can recover the data rows that it does not store or $\mathbf{A}(S_1 \cap C_1^{\complement})$. First, observe that node 1 stores $S_2 \cap \widetilde{C}_1$. Thus, it can recover $\mathbf{A}(S_1 \cap \widetilde{C}_2)$ using part 1 of the message since $\mathbf{A}(S_1 \cap \widetilde{C}_2) = M_1 - \mathbf{A}(S_2 \cap \widetilde{C}_1)$. Similarly, node 1 recovers $\mathbf{A}(S_1 \cap \widetilde{C}_3) = M_2 - \mathbf{A}(S_3 \cap \widetilde{C}_1)$. Finally, from part 4 of the message, node 1 recovers $\mathbf{A}(S_1 \cap \widetilde{C}_{2,3}) = M_4 - \mathbf{A}(S_2 \cap \widetilde{C}_{1,3}) - \mathbf{A}(S_3 \cap \widetilde{C}_{1,2})$.

D. Main Results

We now present the main result of this section, which characterizes the communication rate of the coded scheme. Let $p = \frac{s-q/n}{q-q/n}$.

Theorem 3 (Coded Shuffling Rate): Coded shuffling achieves communication rate

$$R_c = \frac{q}{(np)^2}\left((1-p)^{n+1} + (n-1)p(1-p) - (1-p)^2\right) \quad (18)$$

(in number of data rows transmitted per iteration from the master node), which is significantly smaller than R_u in (17).

The reduction in communication rate is illustrated in Fig. 10 for $n = 50$ and $q = 1000$ as a function of s/q, where $1/n \le s/q \le 1$. For instance, when $s/q = 0.1$, the communication overhead for data-shuffling is reduced by more than 81%. Thus, at a very low storage overhead for caching, the algorithm can be significantly accelerated.

Before we present the proof of the theorem, we briefly compare our main result with similar results shown in [66] and [86]. Our coded shuffling algorithm is related to the coded caching problem [66], since one can design the right cache update rule to reduce the communication rate for an unknown demand or permutation of the data rows. A key difference though is that the coded shuffling algorithm is run over many iterations of the machine learning algorithm. Thus, the right cache update rule is required to guarantee the opportunity of coded transmission at every iteration. Furthermore, the coded shuffling problem has some connections to coded MapReduce [86] as both algorithms mitigate the communication bottlenecks in distributed computation and machine learning. However, coded shuffling enables coded transmission of raw data by leveraging the extra memory space available at each node, while coded MapReduce enables coded transmission of processed data in the shuffling phase of the MapReduce algorithm by cleverly introducing redundancy in the computation of the mappers.

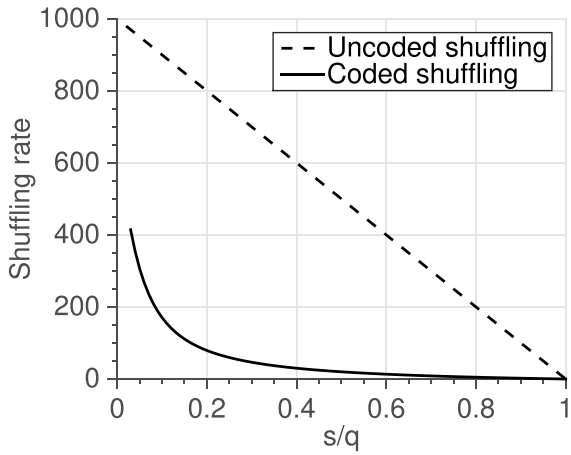


Fig. 10. **The achievable rates of coded and uncoded shuffling schemes.** This figure shows the achievable rates of coded and uncoded schemes versus the cache size for parallel stochastic gradient descent algorithm.

We now prove Theorem 3.

Proof: To find the transmission rate of the coded scheme we first need to find the cardinality of sets $S_i^{t+1} \cap \widetilde{C}_{\mathcal{I}}^t$ for $\mathcal{I} \subset [n]$ and $i \notin \mathcal{I}$. To this end, we first find the probability that a random data row, $\mathbf{r}$, belongs to $\widetilde{C}_{\mathcal{I}}^t$. Denote this probability by $\Pr(\mathbf{r} \in \widetilde{C}_{\mathcal{I}}^t)$. Recall that the cache content distribution at iteration t: q/n rows of cache j are stored with S_j^t and the other $s - q/n$ rows are stored uniformly at random. Thus, we can compute $\Pr(\mathbf{r} \in \widetilde{C}_{\mathcal{I}}^t)$ as follows.

$$\begin{aligned}
&\Pr(\mathbf{r} \in \widetilde{C}_{\mathcal{I}}^t) \\
&= \sum_{i=1}^{n} \Pr(\mathbf{r} \in \widetilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t) \Pr(\mathbf{r} \in S_i^t) && (19)\\
&= \sum_{i=1}^{n} \frac{1}{n} \Pr(\mathbf{r} \in \widetilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t) && (20)\\
&= \sum_{i\in\mathcal{I}} \frac{1}{n} \Pr(\mathbf{r} \in \widetilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t) && (21)\\
&= \sum_{i\in\mathcal{I}} \frac{1}{n} \left(\frac{s-q/n}{q-q/n}\right)^{|\mathcal{I}|-1} \left(1 - \frac{s-q/n}{q-q/n}\right)^{n-|\mathcal{I}|} && (22)\\
&= \frac{|\mathcal{I}|}{n} p^{|\mathcal{I}|-1}(1-p)^{n-|\mathcal{I}|}. && (23)
\end{aligned}$$

(19) is by the law of total probability. (20) is by the fact that $\mathbf{r}$ is chosen randomly. To see (21), note that $\Pr(\mathbf{r} \in \widetilde{C}_{\mathcal{I}}^t | \mathbf{r} \in S_i^t, i \notin \mathcal{I}) = 0$. Thus, the summation can be written only on the indices of $\mathcal{I}$. We now explain (22). Given that $\mathbf{r}$ belongs to S_i^t, and $i \in \mathcal{I}$, then $\mathbf{r} \in C_i$ with probability 1. The other $|\mathcal{I}|-1$ caches with indices in $\mathcal{I}\setminus\{i\}$ contain $\mathbf{r}$ with probability $\frac{s-q/n}{q-q/n}$ independently. Further, the caches with indices in $[n]\setminus\mathcal{I}$ do not contain $\mathbf{r}$ with probability $1-\frac{s-q/n}{q-q/n}$. By defining $p \stackrel{\text{def}}{=} \frac{s-q/n}{q-q/n}$, we have (23).

We now find the cardinality of $S_i^{t+1} \cap \widetilde{C}_{\mathcal{I}}^t$ for $\mathcal{I} \subset [n]$ and $i \notin \mathcal{I}$. Note that $|S_i^{t+1}| = q/n$. Thus, as q gets large (and n remains sub-linear in q), by the law of large numbers,

$$|S_i^{t+1} \cap \widetilde{C}_{\mathcal{I}}^t| = \frac{q}{n} \times \frac{|\mathcal{I}|}{n} p^{|\mathcal{I}|-1}(1-p)^{n-|\mathcal{I}|} + o(q). \quad (24)$$

Recall that for each subset $\mathcal{I}$ with $|\mathcal{I}| \geq 2$, the master node will send $\sum_{i\in\mathcal{I}} \mathbf{A}(S_i \cap \widetilde{C}_{\mathcal{I}\setminus\{i\}})$. Thus, the total rate of coded transmission is

$$R_c = \sum_{i=2}^{n} \binom{n}{i} \frac{q}{n} \frac{i-1}{n} p^{i-2}(1-p)^{n-(i-1)}. \quad (25)$$

To complete the proof, we simplify the above expression. Let $x = \frac{p}{1-p}$. Taking derivative with respect to x from both sides of the equality $\sum_{i=1}^{n} \binom{n}{i} x^{i-1} = \frac{1}{x}\left[(1+x)^n - 1\right]$, we have

$$\sum_{i=2}^{n} \binom{n}{i} (i-1)x^{i-2} = \frac{1+(1+x)^{n-1}(nx-x-1)}{x^2}. \quad (26)$$

Using (26) in (25) completes the proof. ∎

Corollary 1: Consider the case that the cache sizes are just enough to store the data required for processing; that is $s = q/n$. Then, $R_c = \frac{1}{2}R_u$. Thus, one gets a factor 2 reduction gain in communication rate by exploiting coded caching.

Note that when $s = q/n$, $p = 0$. Finding the limit $\lim_{p\to 0} R_c$ in (18), after some manipulations, one calculates

$$R_c = q\left(1-\frac{s}{q}\right)\frac{1}{1+ns/q} = R_u/2, \quad (27)$$

which shows Corollary 1.

Corollary 2: Consider the regime of interest where n, s, and q get large, and $s/q \to c > 0$ and $n/q \to 0$. Then,

$$R_c \to q\left(1-\frac{s}{q}\right)\frac{1}{ns/q} = \frac{R_u}{ns/q} \quad (28)$$

Thus, using coding, the communication rate is reduced by $\Theta(n)$.

Remark 5 (The Advantage of Using Multicasting Over Unicasting):

It is reasonable to assume that $\gamma(n) \simeq n$ for wireless architecture that is of great interest with the emergence of wireless data centers, e.g. [87], [88], and mobile computing platforms [89]. However, still in many applications, the network topology is based on point-to-point communication, and the multicasting opportunity is not fully available, i.e., $\gamma(n) < n$. For these general cases, we have to renormalize the communication cost of coded shuffling since we have assumed that $\gamma(n) = n$ in our results. For instance, in the regime considered in Corollary 2, the renormalized communication cost of coded shuffling R_c^{γ} given $\gamma(n)$ is

$$R_c^{\gamma} = \frac{n}{\gamma(n)} R_c \to \frac{R_u}{\gamma(n)s/q}. \quad (29)$$

Thus, the communication cost of coded shuffling is smaller than uncoded shuffling if $\gamma(n) > q/s$. Note that s/q is the fraction of the data matrix that can be stored at each worker's cache. Thus, in the regime of interest where s/q is a constant independent of n, and $\gamma(n)$ scales with n, the reduction gain of coded shuffling in communication cost is still unbounded and increasing in n.

We emphasize that even in point-to-point communication networks, *multicasting the same message to multiple nodes is significantly faster than unicasting different message (of the same size) to multiple nodes*, i.e., $\gamma(n) \gg 1$, justifying the advantage of using coded shuffling. For instance, the MPI broadcast API (`MPI_Bcast`) utilizes a tree multicast algorithm, which achieves $\gamma(n) = \Theta\left(\frac{n}{\log n}\right)$. Shown in Fig. 11 is the time taken for a data block to be transmitted to an increasing number of workers on an Amazon EC2 cluster, which consists of a point-to-point communication network. We compare the average transmission time taken with MPI scatter (unicast) and that with MPI broadcast. Observe that the average transmission time increases linearly as the number of receivers increases, but with MPI broadcast, the average transmission time increases logarithmically.

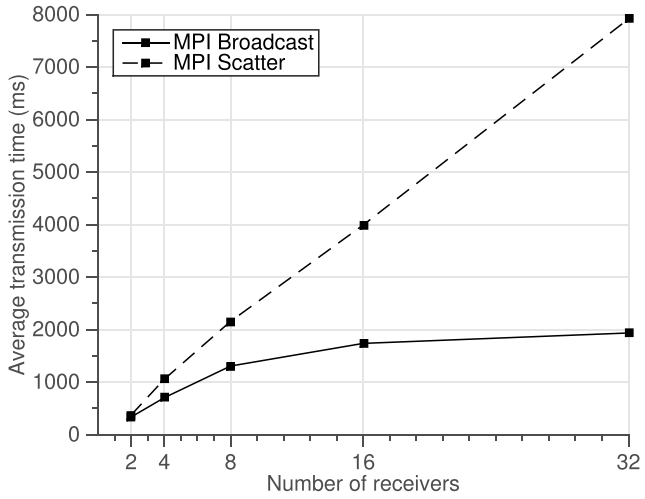


Fig. 11. **Gains of multicasting over unicasting in distributed systems.** We measure the time taken for a data block of size of 4.15 MB to be transmitted to a targeted number of workers on an Amazon EC2 cluster, and compare the average transmission time taken with Message Passing Interface (MPI) scatter (unicast) and that with MPI broadcast. Observe that the average transmission time increases linearly as the number of receivers increases, but with MPI broadcast, the average transmission time increases logarithmically.

V. Conclusion

In this paper, we have explored the power of coding in order to make distributed algorithms robust to a variety of sources of "system noise" such as stragglers and communication bottlenecks. We propose a novel *Coded Computation*

framework that can significantly speed up existing distributed algorithms, by introducing redundancy through codes into the computation. Further, we propose *Coded Shuffling* that can significantly reduce the heavy price of data-shuffling, which is required for achieving high statistical efficiency in distributed machine learning algorithms. Our preliminary experimental results validate the power of our proposed schemes in effectively curtailing the negative effects of system bottlenecks, and attaining significant speedups of up to 40%, compared to the current state-of-the-art methods.

There exists a whole host of theoretical and practical open problems related to the results of this paper. For coded computation, instead of the MDS codes, one could achieve different tradeoffs by employing another class of codes. Then, although matrix multiplication is one of the most basic computational blocks in many analytics, it would be interesting to leverage coding for a broader class of distributed algorithms.

For coded shuffling, convergence analysis of distributed machine learning algorithms under shuffling is not well understood. As we observed in the experiments, shuffling significantly reduces the number of iterations required to achieve a target reliability, but missing is a rigorous analysis that compares the convergence performances of algorithms with shuffling or without shuffling. Further, the trade-offs between bandwidth, storage, and the statistical efficiency of the distributed algorithms are not well understood. Moreover, it is not clear how far our achievable scheme, which achieves a bandwidth reduction gain of $\Theta(\frac{1}{n})$, is from the fundamental limit of communication rate for coded shuffling. Therefore, finding an information-theoretic lower bound on the rate of coded shuffling is another interesting open problem.

REFERENCES

[1] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," presented at the Neural Inf. Process. Syst. Workshop Mach. Learn. Syst., Dec. 2015.

[2] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2016, pp. 1143–1147.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2010, p. 95. [Online]. Available: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Oper. Syst. Design Implement. (OSDI)*, 2004, pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[5] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.

[7] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction," *IEEE Trans. Inf. Theory*, vol. 57, no. 8, pp. 5227–5239, Aug. 2011.

[8] C. Suh and K. Ramchandran, "Exact-repair MDS code construction using interference alignment," *IEEE Trans. Inf. Theory*, vol. 57, no. 3, pp. 1425–1442, Mar. 2011.

[9] I. Tamo, Z. Wang, and J. Bruck, "MDS array codes with optimal rebuilding," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Aug. 2011, pp. 1240–1244.

[10] V. R. Cadambe, C. Huang, S. A. Jafar, and J. Li. (2011). "Optimal repair of MDS codes in distributed storage via subspace interference alignment." [Online]. Available: https://arxiv.org/abs/1106.1250

[11] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe, "Repair optimal erasure codes through Hadamard designs," in *Proc. 49th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, 2011, pp. 1382–1389.

[12] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Trans. Inf. Theory*, vol. 58, no. 11, pp. 6925–6934, Nov. 2011.

[13] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1215–1223.

[14] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple regenerating codes: Network coding for cloud storage," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 2801–2805.

[15] J. Han and L. A. Lastras-Montano, "Reliable memories with subline accesses," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2007, pp. 2531–2535.

[16] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," in *Proc. 6th IEEE Int. Symp. Netw. Comput. Appl. (NCA)*, Jul. 2007, pp. 79–86.

[17] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Mar. 2012, pp. 2771–2775.

[18] G. M. Kamath, N. Prakash, V. Lalitha, and P. V. Kumar. (2012). "Codes with local regeneration." [Online]. Available: https://arxiv.org/abs/1211.1932

[19] A. S. Rawat, O. O. Koyluoglu, N. Silberstein, and S. Vishwanath, "Optimal locally repairable and secure codes for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 60, no. 1, pp. 212–236, Jan. 2014.

[20] N. Prakash, G. M. Kamath, V. Lalitha, and P. V. Kumar, "Optimal linear codes with a local-error-correction property," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2012, pp. 2776–2780.

[21] N. Silberstein, A. S. Rawat and S. Vishwanath, "Error resilience in distributed storage via rank-metric codes," in *Proc. 50th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Monticello, IL, USA, 2012, pp. 1150–1157.

[22] C. Huang *et al.*, "Erasure coding in windows azure storage," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Jun. 2012, pp. 15–26.

[23] M. Sathiamoorthy *et al.*, "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.

[24] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. USENIX HotStorage*, Jun. 2013.

[25] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 331–342.

[26] G. Ananthanarayanan *et al.*, "Reining in the outliers in Map-Reduce clusters using Mantri," in *Proc. 9th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2010, pp. 265–278. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full_papers/Ananthanarayanan.pdf

[27] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 9th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2008, pp. 29–42. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/zaharia/zaharia.pdf

[28] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Proc. 25th Annu. Conf. Neural Inf. Process. Syst. (NIPS)*, 2011, pp. 873–881. [Online]. Available: http://papers.nips.cc/paper/4247-distributed-delayed-stochastic-optimization

[29] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. 25th Annu. Conf. Neural Inf. Process. (NIPS)*, 2011, pp. 693–701.

[30] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2013, pp. 185–198. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan

[31] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" in *Proc. 51st Annu. Allerton Conf. Commun., Control, Comput.*, 2013, pp. 731–738. [Online]. Available: http://dx.doi.org/10.1109/Allerton.2013.6736597

[32] D. Wang, G. Joshi, and G. W. Wornell, "Efficient task replication for fast response times in parallel computation," *ACM SIGMETRICS*, vol. 42, no. 1, pp. 599–600, 2014.

[33] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytiä, "Reducing latency via redundant requests: Exact analysis," *ACM SIGMETRICS*, vol. 43,, no. 1, pp. 347–360, 2015.

[34] M. Chaubey and E. Saule, "Replicated data placement for uncertain scheduling," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop (IPDPS)*, May 2015, pp. 464–472. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2015.50

[35] K. Lee, R. Pedarsani, and K. Ramchandran, "On scheduling redundant requests with cancellation overheads," in *Proc. 53rd Annu. Allerton Conf. Commun., Control, Comput.*, Oct. 2015, pp. 1279–1290.

[36] G. Joshi, E. Soljanin, and G. Wornell, "Efficient redundancy techniques for latency reduction in cloud systems," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 2, pp. 12:1–12:30, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3055281

[37] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes can reduce queueing delay in data centers," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2012, pp. 2766–2770.

[38] K. Lee, N. B. Shah, L. Huang, and K. Ramchandran, "The MDS queue: Analysing the latency performance of erasure codes," *IEEE Trans. Inf. Theory*, vol. 63, no. 5, pp. 2822–2842, May 2017.

[39] G. Joshi, Y. Liu, and E. Soljanin, "On the delay-storage trade-off in content download from coded distributed storage systems," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 989–997, May 2014.

[40] Y. Sun, Z. Zheng, C. E. Koksal, K.-H. Kim, and N. B. Shroff. (2015). "Provably delay efficient data retrieving in storage clouds." [Online]. Available: https://arxiv.org/abs/1501.01661

[41] S. Kadhe, E. Soljanin, and A. Sprintson, "When do the availability codes make the stored data more available?" in *Proc. 53rd Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2015, pp. 956–963.

[42] S. Kadhe, E. Soljanin, and A. Sprintson, "Analyzing the download time of availability codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2015, pp. 1467–1471.

[43] N. Ferdinand and S. Draper, "Anytime coding for distributed computation," presented at the 54th Annu. Allerton Conf. Commun., Control, Comput., Monticello, IL, USA, 2016.

[44] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2092–2100.

[45] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis. (2016). "Gradient coding." [Online]. Available: https://arxiv.org/abs/1612.03301

[46] R. Bitar, P. Parag, and S. E. Rouayheb. (2017). "Minimizing latency for secure distributed computing." [Online]. Available: https://arxiv.org/abs/1703.01504

[47] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2017, pp. 1–2.

[48] A. Reisizadehmobarakeh, S. Prakash, R. Pedarsani, and S. Avestimehr. "Coded computation over heterogeneous clusters." [Online]. Available: https://arxiv.org/abs/1701.05973

[49] K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Coded computation for multicore setups," presened at the ISIT, Jun. 2017.

[50] D. P. Bertsekas, *Nonlinear Programming*. Belmont, MA, USA: Athena Scientific, 1999.

[51] A. Nedic and A. Ozdaglar, "Distributed subgradient methods for multi-agent optimization," *IEEE Trans. Autom. Control*, vol. 54, no. 1, pp. 48–61, Jan. 2009.

[52] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, Jan. 2011.

[53] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[54] J. C. Duchi, A. Agarwal, and M. J. Wainwright, "Dual averaging for distributed optimization: Convergence analysis and network scaling," *IEEE Trans. Autom. Control*, vol. 57, no. 3, pp. 592–606, Mar. 2012.

[55] J. Chen and A. H. Sayed, "Diffusion adaptation strategies for distributed optimization and learning over networks," *IEEE Trans. Signal Process.*, vol. 60, no. 8, pp. 4289–4305, Aug. 2012.

[56] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. 26th Annu. Conf. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1232–1240. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks

[57] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[58] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "MLbase: A distributed machine-learning system," in *Proc. 6th Biennial Conf. Innov. Data Syst. Res. (CIDR)*, Jan. 2013, p. 2. [Online]. Available: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper118.pdf

[59] E. R. Sparks *et al.*, "MLI: An API for distributed machine learning," in *Proc. IEEE 13th Int. Conf. Data Mining (ICDM)*, 2013, pp. 1187–1192. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2013.158

[60] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2014, pp. 583–598. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[61] B. Recht and C. Ré, "Parallel stochastic gradient algorithms for large-scale matrix completion," *Math. Programm. Comput.*, vol. 5, no. 2, pp. 201–226, 2013.

[62] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks Trade*. 2nd ed. 2012, pp. 421–436. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35289-8_25

[63] C. Zhang and C. Ré, "Dimmwitted: A study of main-memory statistical analytics," *Proc. VLDB Endowment*, vol. 7, no. 12, pp. 1283–1294, 2014.

[64] M. Gürbüzbalaban, A. Ozdaglar, and P. Parrilo. (2015). "Why random reshuffling beats stochastic gradient descent." [Online]. Available: https://arxiv.org/abs/1510.08560

[65] S. Ioffe and C. Szegedy. (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift," [Online]. Available: https://arxiv.org/abs/1502.03167

[66] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," *IEEE Trans. Inf. Theory*, vol. 60, no. 5, pp. 2856–2867, May 2014.

[67] M. A. Maddah-Ali and U. Niesen, "Decentralized coded caching attains order-optimal memory-rate tradeoff," *IEEE/ACM Trans. Netw.*, vol. 23, no. 4, pp. 1029–1040, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1109/TNET.2014.2317316

[68] R. Pedarsani, M. A. Maddah-Ali, and U. Niesen, "Online coded caching," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2014, pp. 1878–1883. [Online]. Available: http://dx.doi.org/10.1109/ICC.2014.6883597

[69] N. Karamchandani, U. Niesen, M. A. Maddah-Ali, and S. Diggavi, "Hierarchical coded caching," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2014, pp. 2142–2146.

[70] M. Ji, G. Caire, and A. F. Molisch, "Fundamental limits of distributed caching in D2D wireless networks," in *Proc. IEEE Inf. Theory Workshop (ITW)*, Sep. 2013, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1109/ITW.2013.6691247

[71] S. Li, M. A. Maddah-ali, and S. Avestimehr, "Coded MapReduce," presented at the 53rd Annu. Allerton Conf. Commun., Control, Comput., Monticello, IL, USA, 2015.

[72] Y. Birk and T. Kol, "Coding on demand by an informed source (ISCOD) for efficient broadcast of different supplemental data to caching clients," *IEEE Trans. Inf. Theory*, vol. 52, no. 6, pp. 2825–2830, Jun. 2006.

[73] Z. Bar-Yossef, Y. Birk, T. S. Jayram, and T. Kol, "Index coding with side information," *IEEE Trans. Inf. Theory*, vol. 57, no. 3, pp. 1479–1494, Mar. 2011.

[74] M. A. Attia and R. Tandon, "Information theoretic limits of data shuffling for distributed learning," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[75] M. A. Attia and R. Tandon, "On the worst-case communication overhead for distributed data shuffling," in *Proc. 54th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2016, pp. 961–968.

[76] L. Song and C. Fragouli. (2017). "A pliable index coding approach to data shuffling." [Online]. Available: https://arxiv.org/abs/1701.05540

[77] S. Li, M. A. Maddah-Ali and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, Washington, DC, USA, 2016, pp. 1–6.

[78] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Hoboken, NJ, USA: Wiley, 2012.

[79] S. Lin and D. J. Costello, *Error Control Coding*, vol. 2. Englewood Cliffs, NJ, USA: Prentice-Hall, 2004.

[80] G. Liang and U. C. Kozat, "TOFEC: Achieving optimal throughput-delay trade-off of cloud storage using erasure codes," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2014, pp. 826–834. [Online]. Available: http://dx.doi.org/10.1109/INFOCOM.2014.6848010

[81] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[82] X. Meng *et al.*, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, 2016.

[83] *Open MPI: Open Source High Performance Computing*. Accessed on Nov. 25, 2015. [Online]. Available: http://www.open-mpi.org
[84] *StarCluster*. Accessed on Nov. 25, 2015. [Online]. Available: http://star.mit.edu/cluster/
[85] *BLAS (Basic Linear Algebra Subprograms)*. Accessed on Nov. 25, 2015. [Online]. Available:http://www.netlib.org/blas/
[86] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Fundamental tradeoff between computation and communication in distributed computing," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2016, pp. 1814–1818.
[87] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting data center networks with multi-gigabit wireless links," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 38–49, Aug. 2011.
[88] Y. Zhu *et al.*, "Cutting the cord: A robust wireless facilities network for data centers," in *Proc. 20th Annu. Int. Conf. Mobile Comput. Netw.*, 2014, pp. 581–592.
[89] M. Y. Arslan, I. Singh, S. Singh, H. V. Madhyastha, K. Sundaresan, and S. V. Krishnamurthy, "Computing while charging: Building a distributed computing infrastructure using smartphones," in *Proc. 8th Int. Conf. Emerg. Netw. Experim. Technol.*, 2012, pp. 193–204.

Kangwook Lee is a postdoctoral researcher in the School of Electrical Engineering, KAIST. Kangwook earned his Ph.D. in EECS from UC Berkeley in 2016, under the supervision of Kannan Ramchandran. He is a recipient of the KFAS Fellowship from 2010 to 2015. His research interests lie in information theory and machine learning.

Maximilian Lam is a computer science student at UC Berkeley whose main research interests are systems and machine learning.

Ramtin Pedarsani is an Assistant Professor in ECE Department at the University of California, Santa Barbara. He received the B.Sc. degree in electrical engineering from the University of Tehran, Tehran, Iran, in 2009, the M.Sc. degree in communication systems from the Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, in 2011, and his Ph.D. from the University of California, Berkeley, in 2015. His research interests include networks, machine learning, stochastic systems, information and coding theory, and transportation systems. Ramtin is a recipient of the IEEE international conference on communications (ICC) best paper award in 2014.

Dimitris Papailiopoulos is an Assistant Professor of Electrical and Computer Engineering at the University of Wisconsin-Madison and a Faculty Fellow of the Grainger Institute for Engineering. Between 2014 and 2016, Papailiopoulos was a postdoctoral researcher in EECS at UC Berkeley and a member of the AMPLab. His research interests span machine learning, coding theory, and distributed algorithms, with a current focus on coordination-avoiding parallel machine learning and the use of erasure codes to speed up distributed computation. Dimitris earned his Ph.D. in ECE from UT Austin in 2014, under the supervision of Alex Dimakis. In 2015, he received the IEEE Signal Processing Society, Young Author Best Paper Award.

Kannan Ramchandran (Ph.D.: Columbia University, 1993) is a Professor of Electrical Engineering and Computer Sciences at UC Berkeley, where he has been since 1999. He was on the faculty at the University of Illinois at Urbana-Champaign from 1993 to 1999, and with AT&T Bell Labs from 1984 to 1990. He is an IEEE Fellow, and a recipient of the 2017 IEEE Kobayashi Computers and Communications Award, which recognizes outstanding contributions to the integration of computers and communications. His research awards include an IEEE Information Theory Society and Communication Society Joint Best Paper award for 2012, an IEEE Communication Society Data Storage Best Paper award in 2010, two Best Paper awards from the IEEE Signal Processing Society in 1993 and 1999, an Okawa Foundation Prize for outstanding research at Berkeley in 2001, an Outstanding Teaching Award at Berkeley in 2009, and a Hank Magnuski Scholar award at Illinois in 1998. His research interests are at the intersection of signal processing, coding theory, communications and networking with a focus on theory and algorithms for large-scale distributed systems.