Speeding Up Distributed Machine Learning Using Codes

Kangwook Lee[ϵ], Maximilian Lam[ϵ], Ramtin Pedarsani[ϵ],
Dimitris Papailiopoulos[α,ϵ] and Kannan Ramchandran[ϵ]
[α]AMPLab, [ϵ]EECS at UC Berkeley*

December 11, 2015

Abstract

Codes are widely used in many engineering applications to offer some form of reliability and fault tolerance. The high-level idea of coding is to exploit resource redundancy to deliver higher *robustness* against *system 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. Moreover, redundancy is abundant: a plethora of nodes, a lot of spare storage, etc. However, there has been little interaction cutting across *Codes*, *Machine Learning*, and *Distributed Systems*.

In this work, scratching the surface of "codes for distributed computation," we provide theoretical insights on how *coded* solutions can achieve significant gains compared to 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 leverage the plethora of nodes and alleviate the effects of stragglers. We show that 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. In data shuffling, we use codes to exploit the excess in storage and reduce communication bottlenecks. We show that when α is the fraction of the data matrix that can be cached at each worker, and n is the number of workers, *coded shuffling* reduces the communication cost by a factor $\Theta(\alpha\gamma(n))$ compared to uncoded shuffling, where $\gamma(n)$ is the ratio of the cost of unicasting n messages to n users to broadcasting a common message (of the same size) to n users. For instance, $\gamma(n) \simeq n$ if broadcasting a message to n users is as cheap as unicasting a message to one user.

We provide evidence on synthetic and Open MPI experiments on Amazon EC2 that highlights significant gains offered by our proposed *coded* solutions compared to uncoded ones: our preliminary results show that *coded* distributed algorithms can achieve significant speedups of up to 40% compared to *uncoded* distributed algorithms.

1 Introduction

In recent years, the computational paradigm for large-scale machine learning and data analytics has started to move towards massively large distributed systems, comprising individually small and unreliable computational nodes (low-end, commodity hardware). Specifically, modern distributed systems like Apache Spark [1] and computational primitives like MapReduce [2] have gained significant traction, as they enable the execution of production-scale tasks on data sizes of the order of terabytes. The backbone of these large and complex platforms consists of three functional layers: (i) a computational layer; (ii) a communication layer (to move data around the system as needed); and (iii) a storage layer. 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. Moreover, given the individually unpredictable nature of the computational nodes in these systems, we are faced with the challenge of securing fast and high-quality algorithmic results in the face of uncertainty. This, coupled with the high level of complexity and heterogeneity of the component hardware, introduces significant *delays* that represent a key bottleneck to attaining the promised speed-ups of these large systems.

In this work, we view distributed machine learning through a robust coding-theory lens. 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.). Likewise, codes have begun

*Emails: kw1jjang@eecs.berkeley.edu, agnusmaximus@berkeley.edu, ramtin@eecs.berkeley.edu, dimitrisp@berkeley.edu, kannanr@eecs.berkeley.edu.

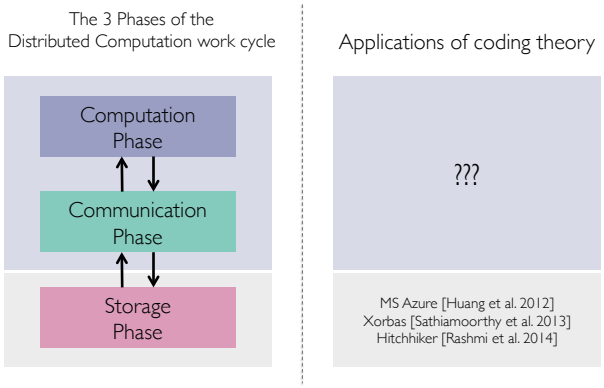


Figure 1: **Conceptual diagram of the work-flow of distributed algorithms.** It is helpful to abstract the algorithmic work-flow of distributed tasks as operating in three distinct phases: a computation, a communication, and a storage phase. We are driven by this abstraction and our expertise in coding solutions for storage, to optimize the communication and local computation phases using codes.

to transform the evolution of large-scale distributed storage systems in modern data centers under the umbrella of regenerating and locally repairable codes for distributed storage [3–18] which are also having a major impact on industry [19–22].

We envision codes to play a similar transformational role in the way next-generation large-scale computational platforms are deployed. Since the performance of distributed algorithmic solutions is significantly affected by anomalous system behavior and bottlenecks [23], i.e., a form of "system noise", there is an exciting opportunity for codes to endow distributed platforms with *robustness* against system noise. This forms the key motivation of this paper which is driven by the fundamental question:

Can we use codes to guarantee robust speedups in distributed computation?

In addressing our main question, we observe that in modern large-scale systems, there is a surplus of resource *redundancy* that can be exploited: a plethora of nodes, and an abundance of storage. Can codes exploit these resources to speed up computation? We address this by decomposing distributed computation into three functional phases, and highlight opportunities where coding can be useful. As stated earlier, we can abstract a distributed algorithmic workflow as operating in three distinct phases: a storage, a communication, and a computation phase (see Fig 1). In this paper, we use coding theory and focus on improving the bottlenecks caused during *the communication and computation phases* of distributed algorithms. 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 leverage the plethora of nodes and alleviate the effects of stragglers. We show that 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. In data shuffling, we use codes to exploit the excess in storage and reduce communication bottlenecks. We show that when α is the fraction of the data matrix that can be cached at each worker, and n is the number of workers, *coded shuffling* reduces the communication cost by a factor $\Theta(\alpha\gamma(n))$ compared to uncoded shuffling, where $\gamma(n)$ is the ratio of the cost of unicasting n messages to n users to broadcasting a common message (of the same size) to n users. For instance, $\gamma(n) \simeq n$ if broadcasting a message to n users is as cheap as unicasting a message to one user.

We provide evidence on synthetic and Open MPI experiments on Amazon EC2 that highlights significant gains offered by our proposed *coded* solutions compared to uncoded ones: our preliminary results show that *coded* distributed algorithms can achieve significant speedups of up to 40% compared to *uncoded* distributed algorithms.

1.1 Main Ideas

The archetypal goal of distributed computation is to efficiently compute a function by scaling out across a number of machines, while maximizing the speedup gained by having k machines versus that of serial computation. We will consider potentially iterative algorithms that are commonly deployed: the computed function value can be potentially fed back to the input of the system for the subsequent computations. The three phases of distributed computation, as described before, are shown in Fig. 2, where the Δ boxes indicate that there are potential delays (e.g., caused by

communication bottlenecks, system failures, and/or slow nodes).

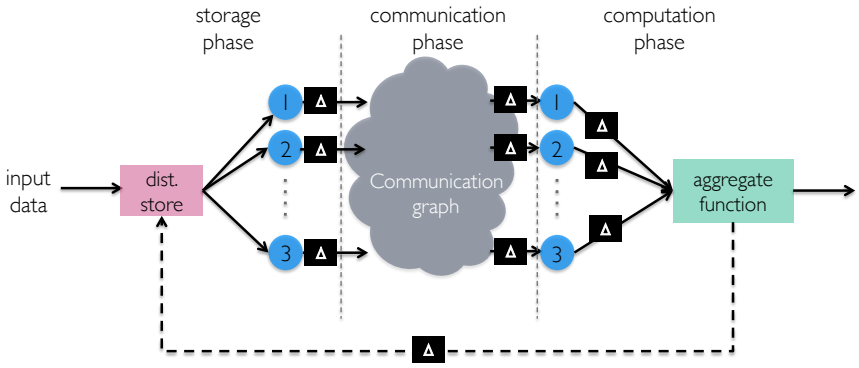


Figure 2: **Conceptual diagram of the phases of distributed computation.** The algorithmic workflow of distributed (potentially iterative) tasks, can be seen as receiving input data, storing them in distributed nodes, communicating data around the distributed network, and then computing locally a function 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.

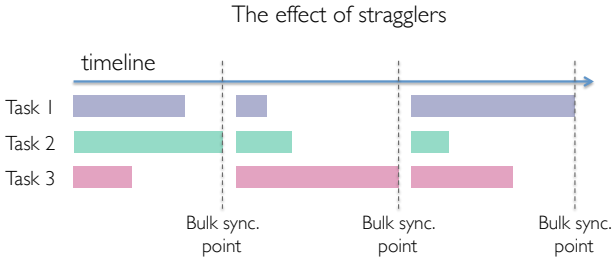


Figure 3: **The effects of slow nodes.** In distributed computation, the running time of a single distributed task is governed by that of the slowest node. In this toy figure, we see how slow nodes can significantly impact the running time of distributed computation. Can we use coding to alleviate the straggler's effects?

In order to maximize system performance, we need to identify and accelerate the core building blocks of distributed algorithms. We identify two basic 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: *computing linear functions* and *data shuffling*. We highlight these two candidate applications where applying ideas from coding theory can significantly improve the overall running time of distributed algorithms, by minimizing delays and overcoming bottlenecks.

We first propose **coded computation** in Section 3, where we use codes to speed up distributed algorithms that perform linear operations. We do so by leveraging the abundance of nodes to mitigate the effects of stragglers (i.e., nodes that are significantly slower than average as show in Fig. 3), and guarantee a fast and seamless execution that is robust against worst-case delays. We then propose a novel **coded data-shuffling** framework in Section 5 that can provably alleviate the communication bottleneck when performing data shuffles (i.e., a random permutation and allocation of data-points across nodes). Data shuffling is a core element of many machine learning applications, and well-known to improve the statistical performance of learning algorithms. For this application, we exploit any excess storage available to store some cached information per node that can be used to significantly reduce communication during data shuffling across different iterations.

We would like to remark that a major innovation of our coding solutions is that they are woven into the fabric of the algorithm 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).

1.2 Related Prior Work

1.2.1 Coded Computation and Straggler Mitigation

The straggler problem has been widely observed in distributed computing clusters. The authors of [23] 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. The authors of [24] characterize the impacts 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. In [25], Zaharia et al. propose a modification to the existing straggler detection algorithm and show that the proposed solution can effectively reduce the completion time of MapReduce tasks. In [24], Ananthanarayanan et al. 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 promising approaches is based on appropriate modification of the algorithms. That is, one can design distributed algorithms that are robust to *asynchronous* or delayed updates from the workers. Such robust distributed algorithms can continuously make progress without needing to wait for all the responses from the workers, and hence the overall runtime of these algorithms is less affected by stragglers. For instance, the authors of [26] study the convergence of asynchronous stochastic gradient descent (SGD) algorithms, and show that their convergence rate is order-optimal. Moreover, in the single-node multi-core setup the authors in [27] propose HOGWILD! an asynchronous multicore implemnentation fo SGD, that runs without any memory locking and synchronization mechanisms between multiple threads. We would like to note that HOGWILD! and other asynchronous approaches do not in general guarantee "correctness" of the result, i.e., the output of the asynchronous algorithm can differ from that of a serial execution with an identical number of iterations; this may not be an issue in statistical problems, as the end solution will be noisy, but can become critically important if one wishes to have a high-precision solution, e.g., as is the case for exact matrix multiplication.

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 [28–33]. 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. In [33], the authors 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 Section 3, 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).

1.2.2 Data Shuffling and Communication Overheads

Distributed learning algorithms on large-scale networked systems have been extensively studied in the literature [34–44]. 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 [27, 45–49]. 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 [45, 48, 49]—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 build upon the coded caching scheme by Maddah-Ali and Niesen [50]. 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 [51], online coded caching [52], hierarchical coded caching for wireless communication [53], and device-to-device coded caching [54]. Recently, [55] 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 ML 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 Section 5, 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.

We would also like to remark that there has been significant work in communication avoiding algorithms in the context of parallel numerical analysis and linear algebra [56–59]. In contrast to this line of work, we propose the use of coding opportunities to strike a balance between statistical efficiency due to shuffling and the cost of communicating data points across different data passes.

2 Overview of Main Results

In this section, we provide a brief overview of the main results of this paper.

2.1 Coded Computation

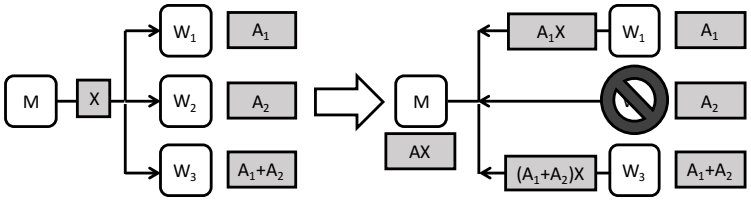


Figure 4: **Illustration of *Coded Matrix Multiplication*.** Data matrix $\mathbf{A}$ is partitioned into 2 submatrices: $\mathbf{A}_1$ and $\mathbf{A}_2$. Node W_1 stores $\mathbf{A}_1$, node W_2 stores $\mathbf{A}_2$, and node W_3 stores $\mathbf{A}_1 + \mathbf{A}_2$. Upon receiving $\mathbf{X}$, each 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.

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. 4. The goal is to compute a matrix multiplication $\mathbf{AX}$. The data matrix $\mathbf{A}$ is vertically divided into two (equally tall) submatrices $\mathbf{A}_1$ and $\mathbf{A}_2$, which are stored in node 1 and node 2, respectively. In node 3, we store the sum of the two submatrices $\mathbf{A}_1 + \mathbf{A}_2$. 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. For instance, consider a case where it collects $\mathbf{A}_1\mathbf{X}$ from node 1 and $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$ from node 3. 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 $\mathbf{AX}$, which is a vertical concatenation of $\mathbf{A}_1\mathbf{X}$ and $\mathbf{A}_2\mathbf{X}$.

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 exemplar 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 proofs, see Section 3.5.

2.2 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.

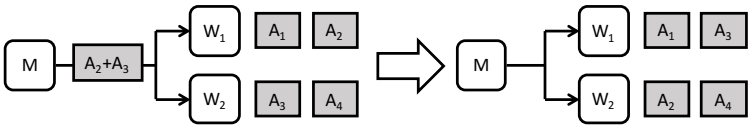


Figure 5: **Illustration of *Coded Shuffling*.** Data matrix $\mathbf{A}$ is partitioned into 4 submatrices: $\mathbf{A}_1$ to $\mathbf{A}_4$. Before shuffling, worker W_1 has $\mathbf{A}_1$ and $\mathbf{A}_2$ and worker W_2 has $\mathbf{A}_3$ and $\mathbf{A}_4$. The master node can send $\mathbf{A}_2 + \mathbf{A}_3$ in order to shuffle the data stored at the two workers.

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 master node holds the entire data set $\mathbf{A}$, a set of n datapoints. For the sake of this example, assume the data to be equipartitioned in 4 batches $\mathbf{A}_1, \ldots, \mathbf{A}_4$. Assume that node W_1 already has $\mathbf{A}_1$ and $\mathbf{A}_2$ cached, and node W_2 has $\mathbf{A}_3$ and $\mathbf{A}_4$ cached. Moreover, assume that the sole objective of the master is to transmit to the first worker $\mathbf{A}_3$ and to the second $\mathbf{A}_4$. For this purpose, the master node can simply broadcast a *coded* message $\mathbf{A}_2 + \mathbf{A}_3$ to the worker nodes. Since node 1 has access to $\mathbf{A}_2$, it can subtract $\mathbf{A}_2$ from the received message $\mathbf{A}_2 + \mathbf{A}_3$, and replace $\mathbf{A}_2$ with $\mathbf{A}_3$. Similarly, node 2 can replace $\mathbf{A}_3$ with $\mathbf{A}_2$. 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. This is true assuming that *broadcasting* a message to all workers is significantly cheaper than sending individual messages to each worker. More formally, let $\gamma(n)$ be the ratio of the cost of unicasting n messages to n users to broadcasting 1 message (of the same size) to n users. Clearly, $1 \leq \gamma(n) \leq n$, where $\gamma(n) \simeq n$ corresponds to the case that broadcasting a message is as cheap as unicasting one, and $\gamma(n) \simeq 1$ corresponds to the case that there is almost no gain in broadcasting a message as it is as costly as unicasting it individually to the n worker nodes.

The *Coded Shuffling* algorithm is a generalization of the above toy example, which we explain in detail in Section 5. The main result on *Coded Shuffling* is stated 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. Coded shuffling reduces the communication cost by a factor $\Theta(\alpha\gamma(n))$ compared to uncoded shuffling.*

For the formal version of the theorem and its proofs, see Section 5.3.

3 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.

3.1 Erasure Codes

An erasure code is a method of introducing redundancy to a message that needs to be protected against noise such as erasures in telecommunication channels, packet drops in the routing layer of the Internet, and disk failures in data storage systems [60]. An erasure code *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 [60, 61].

An important class of erasure codes is the class of *repetition codes*. Given k message symbols, an $\frac{n}{k}$-repetition code simply repeats each symbol $\frac{n}{k}$ times. Thus, one can recover the original message as long as at least one of the $\frac{n}{k}$ repeated symbols is not erased for each of the k message symbols. Due to its simplicity, repetition codes have been widely used in many applications including modern distributed storage systems.

Another important class of codes is Maximum-Distance Separable (MDS) codes. A well-known example is the Reed-Solomon code used to protect CDs and DVDs. When a message is encoded using an (n, k) MDS code, any set of k out of the n encoded symbols is sufficient to recover the message of k symbols. As a concrete example, consider a message of two real numbers $\mathbf{m} = (m_A, m_B) \in \mathbb{R}^2$. Consider a code that transforms $\mathbf{m}$ into $\mathbf{c} =$

$(m_A, m_B, m_A + m_B)$. Clearly, the original message $\mathbf{m}$ can be recovered with any $k = 2$ symbols of $\mathbf{c}$, so the code is an $(n = 3, k = 2)$ MDS code.

3.2 Coded Computation

We now formally define coded computation.

Definition 1 (Coded computation). Consider a computational task $f_A(\cdot)$. A *coded* distributed algorithm for computing $f_A(\cdot)$ is specified by

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

where $[n] = \{1, 2, \ldots, n\}$ and $\mathcal{P}(\cdot)$ is the powerset 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 $f_A(x)$ 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 A_i is stored at the i^{th} worker for all i. Upon receiving the input argument x, the master node broadcasts 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 $f_A(x)$. Algorithms 1 and 2 summarize the described protocols of the master node and the worker nodes.

Algorithm 1 Coded computation: master node's protocol

on Receiving an input argument x
  Broadcast x to all the workers.
  $\mathbf{i} = \langle\rangle$
  $y_{list} = \langle\rangle$
  while $\mathbf{i} \notin \mathcal{I}$ **do**
    on Receiving a message y from worker j
      $\mathbf{i} \leftarrow \langle \mathbf{i}, j\rangle$
      $y_{list} \leftarrow \langle y_{list}, y\rangle$
  end while
  $y \leftarrow \texttt{dec}(\mathbf{i}, y_{list})$
  Return y

Algorithm 2 Coded computation: worker node i's protocol

on Receiving an input argument x
  Compute $y_i = f^i_{A_i}(x)$
  Send y_i to the master node

The algorithm described in Section 2 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 A is first divided into k (equally tall) submatrices [1]. Then, by applying an (n, k) MDS code to each element of the submatrices, n encoded submatrices are obtained. Upon receiving *any* k task results, the master node can use the decoding algorithm to decode k task results. Then, one can find Ax simply by concatenating them.

[1] If the number of rows of A is not a multiple of k, one can append zero rows to A to make the number of rows a multiple of k.

3.3 Runtime of Uncoded/Coded Distributed Algorithms

In this section, we analyze runtime of uncoded and coded distributed algorithms. We first consider the overall runtime of an uncoded distributed algorithm, $T_{\text{overall}}^{\text{uncoded}}$. Assuming that the runtime of each task is identically distributed and independent of others, and denoting the runtime of the i^{th} worker by T^i,

$$T_{\text{overall}}^{\text{uncoded}} = T_{(n)} \stackrel{\text{def}}{=} \max\{T^1, T^2, \ldots, T^n\}, \tag{1}$$

where $T_{(i)}$ is the i^{th} smallest one in $\{T^i\}_{i=1}^n$. From (1), 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.,

$$T_{\text{overall}}^{\text{coded}} = T_{(\mathcal{I})} \stackrel{\text{def}}{=} \min_{\mathbf{i}\in\mathcal{I}} \max_{j\in\mathbf{i}} T_j \tag{2}$$

We remark that the runtime of uncoded distributed algorithms (1) is a special case of (2) 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 are all the minimal sets that have k distinct task results, i.e., $\mathcal{I} = \prod_{i=1}^k \{(i-1)\frac{n}{k}+1, (i-1)\frac{n}{k}+2, \ldots, i\frac{n}{k}\}$. Thus,

$$T_{\text{overall}}^{\text{Repetition-coded}} = \max_{i\in[k]} \min_{j\in[\frac{n}{k}]} \{T^{(i-1)\frac{n}{k}+j}\}. \tag{3}$$

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., $\mathcal{I} = \{\mathbf{i} | \mathbf{i} \in [n],\ |\mathbf{i}| = k\}$. Thus,

$$T_{\text{overall}}^{\text{MDS-coded}} = T_{(k)} \tag{4}$$

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

3.4 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 an appropriate 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 positive-valued, continuous random variable distributed according to F, i.e. $\Pr(T_0 \le t) = F(t)$. 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., $\Pr(T^i \le t) = F(\ell t)$ for $1 \le i \le \ell$. Finally, the computing times of the k tasks are assumed to be independent of one another.

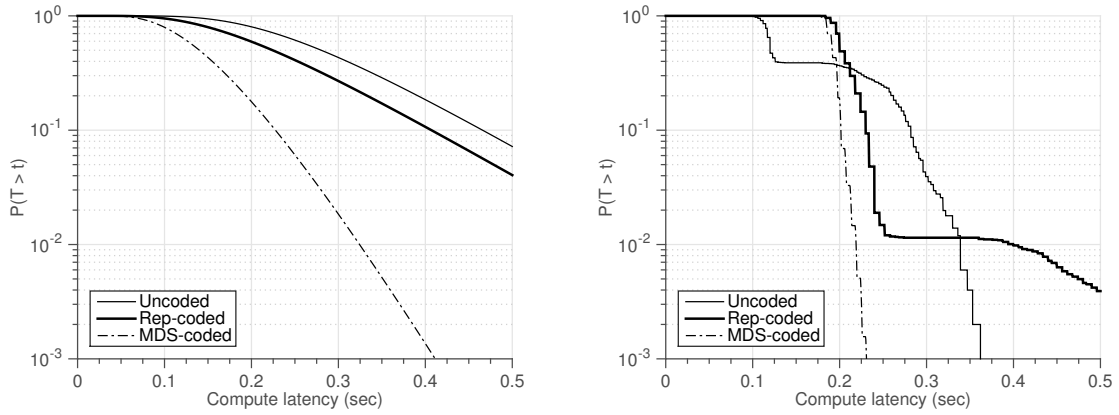
We first consider an uncoded distributed algorithm with n (uncoded) subtasks. Based on the assumptions mentioned above, the runtime of each subtask is $F(nt)$. Thus, the runtime distribution of an uncoded distributed algorithm, denoted by $F_{\text{overall}}^{\text{uncoded}}(t)$, is simply $[F(nt)]^n$.

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 (3) and (4), 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 $\min_{j\in[\frac{n}{k}]}\{T^{(i-1)\frac{n}{k}+j}\}$, and then find the distribution of the maximum of k such terms:

$$F_{\text{overall}}^{\text{Repetition}}(t) = \left[1 - [1 - F(kt)]^{\frac{n}{k}}\right]^k. \tag{5}$$

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

$$F_{\text{overall}}^{\text{MDS-coded}}(t) = \int_{\tau=0}^{t} nkf(k\tau)\binom{n-1}{k-1} F(k\tau)^{k-1} [1 - F(k\tau)]^{n-k}\, d\tau. \tag{6}$$



(a) Shifted-exponential distribution

(b) Empirical distribution

Figure 6: **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. In (b), we use the empirical task runtime distribution measured on an Amazon EC2 cluster.

Remark 1. For the same value 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 is strictly larger than that of the repetition-coded algorithm.

In Fig. 6, 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 $k = 5$. For (a), we use a shifted-exponential distribution as the mother runtime distribution. That is, $F(t) = 1 - e^{t-1}$ for $t \geq 1$. For (b), we use the empirical task runtime distribution that is measured on an Amazon EC2 cluster [2]. Observe that for both cases, the runtime distribution of the MDS-coded distribution has the lightest tail.

3.5 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 at first 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 easily compute the overall runtime distribution of the coded distributed algorithm using (5) and (6). Then, one can optimize the design based on various target metrics, e.g., the expected overall runtime, the 99^{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 [62]: the authors used this distribution to model latency of file queries from cloud storage systems. The shifted-exponential distribution is the sum of a constant and an exponential random variable, i.e.,

$$\Pr(T_0 \leq t) = 1 - e^{-\mu(t-1)}, \ \ \forall t \geq 1, \tag{7}$$

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

With this shifted-exponential model, we first find exact, closed-form expressions for 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

[2] The detailed description of the experiments is provided in Section 4.

is

$$\mathbb{E}[T_{\text{overall}}^{\text{uncoded}}] = \frac{1}{n}\left(1 + \frac{1}{\mu}\log n\right) = \Theta\left(\frac{\log n}{n}\right). \tag{8}$$

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,

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

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,

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

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 3 (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* not *replicating tasks or setting $k = n$. 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 (9) as a function of k has a unique extreme point. By differentiating (9) 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 $\mu < 0$; however, the optimal repetition-coded distributed algorithm still suffers from the factor of $\mathcal{O}(\log n)$, 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 4 (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*

$$k^\star = \left[1 + \frac{1}{W_{-1}(-e^{-\mu-1})}\right] n \stackrel{\text{def}}{=} \alpha^\star(\mu) n, \tag{11}$$

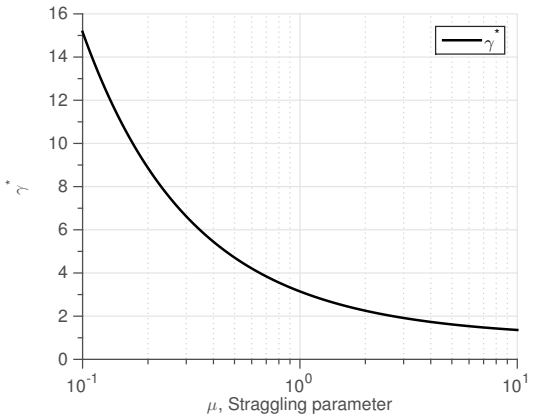
and $W_{-1}(\cdot)$ is the lower branch of Lambert W function [3] *Thus,*

$$\min_k \mathbb{E}[T_{\text{overall}}^{\text{MDS-coded}}] = \frac{-W_{-1}(-e^{-\mu-1})}{\mu n} \stackrel{\text{def}}{=} \frac{\gamma^\star(\mu)}{n}. \tag{12}$$

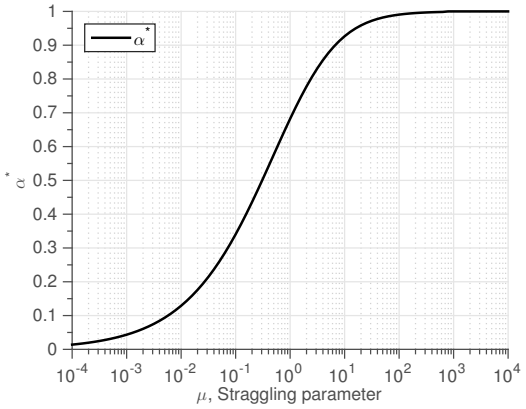
Proof. It is easy to see that (10) as a function of k has a unique extreme point. By differentiating (10) with respect to k and equating it to zero, we have $\frac{1}{k^\star}\left(1 + \frac{1}{\mu}\log\left(\frac{n}{n-k^\star}\right)\right) = \frac{1}{\mu}\frac{1}{n-k^\star}$. By setting $k = \alpha^\star n$, 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 $\gamma^*(\mu)$ and $\alpha^\star(\mu)$ in Fig. 7. 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 $\frac{\gamma^\star(\mu)}{n} \lesssim \frac{3.15}{n}$, which is only 3.15 away from the lower bound.

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



(a) $\gamma^\star$ in Lemma 4 as a function of μ.

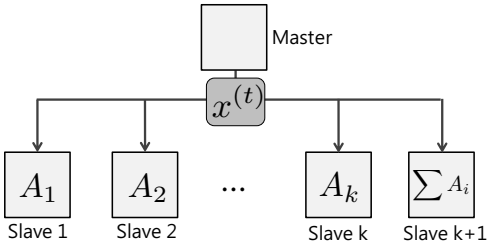


(b) $\alpha^\star$ in Lemma 4 as a function of μ.

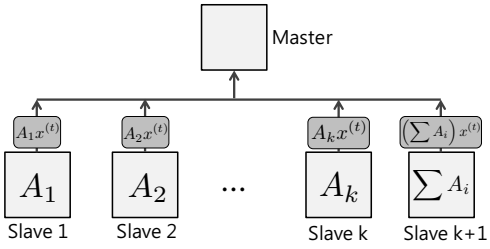
Figure 7: $\gamma^\star$ **and** $\alpha^\star$ **in Lemma 4.** As a function of the straggling parameter, we plot γ^* and α^*, which help quantify the runtime overhead of the straggler problem and design the optimal MDS-coded computation, respectively.

Remark 2 (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 Section 3.2, 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 $\frac{n}{k^\star} - 1 = \frac{1}{\alpha^\star} - 1$.

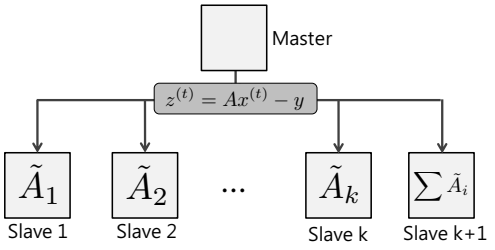
3.6 Coded Gradient Descent: An MDS-coded Distributed Algorithm for Linear Regression



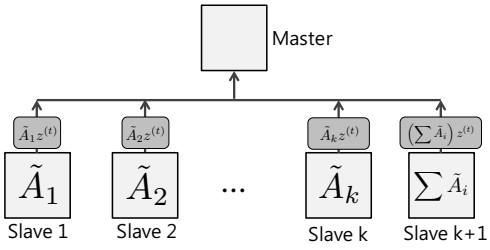
(a) In the beginning of the t^{th} iteration, the master node broadcasts $x^{(t)}$ to the worker nodes.



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



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



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

Figure 8: **Illustration of the coded gradient descent 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 $Ax^{(t)}$ as depicted in (a) and (b), and then computes $A^T(Ax^{(t)} - y)$ as depicted in (c) and (d).

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,

$$\min_x f(x) \stackrel{\text{def}}{=} \min_x \frac{1}{2}\|Ax - y\|_2^2, \tag{13}$$

where $y \in \mathbb{R}^q$ is the label vector, $A = [a_1, a_2, \ldots, a_n]^T \in \mathbb{R}^{q \times r}$ is the data matrix, and $x \in \mathbb{R}^r$ is the unknown weight vector to be found. We seek a distributed algorithm to solve this regression problem. Since $f(x)$ is convex in x, the gradient-based distributed algorithm works as follows. We first compute the objective function's gradient: $\nabla f(x) = A^T(Ax - y)$. Denoting by $x^{(t)}$ the estimate of x in the t^{th} iteration, we iteratively update $x^{(t)}$ according to the following equation.

$$x^{(t+1)} = x^{(t)} - \gamma \nabla f(x^{(t)}) = x^{(t)} - \gamma A^T(Ax^{(t)} - y) \tag{14}$$

The above algorithm is guaranteed to converge to the optimal solution if we use a small enough step size γ [63], 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` [64]. As $A^T(Ax^{(t)} - y) = \sum_{i=1}^q a_i(a_i^T x^{(t)} - y_i)$, 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. 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.

$$Ax^{(t)}, \;\; A^T(Ax^{(t)} - y) \stackrel{\text{def}}{=} A^T z^{(t)} \tag{15}$$

In Section 3.2, 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 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 A as A_i ($\tilde{A}_i$), the i^{th} worker stores both A_i and $\tilde{A}_i$. In the beginning of each iteration, the master node broadcasts $x^{(t)}$ to the worker nodes, each of which computes the local matrix multiplication for $Ax^{(t)}$ 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 $z^{(t)} = Ax^{(t)}$. The master node now broadcasts $z^{(t)}$ to the workers, and the workers compute local matrix multiplication for $A^T z^{(t)}$. Finally, the master node can decode $A^T z^{(t)}$ as soon as it receives any k_2 task results, and can proceed to the next iteration. Fig. 8 illustrates the protocol with $k_1 = k_2 = n - 1$.

Remark 3 (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 $k_2 \simeq n$.

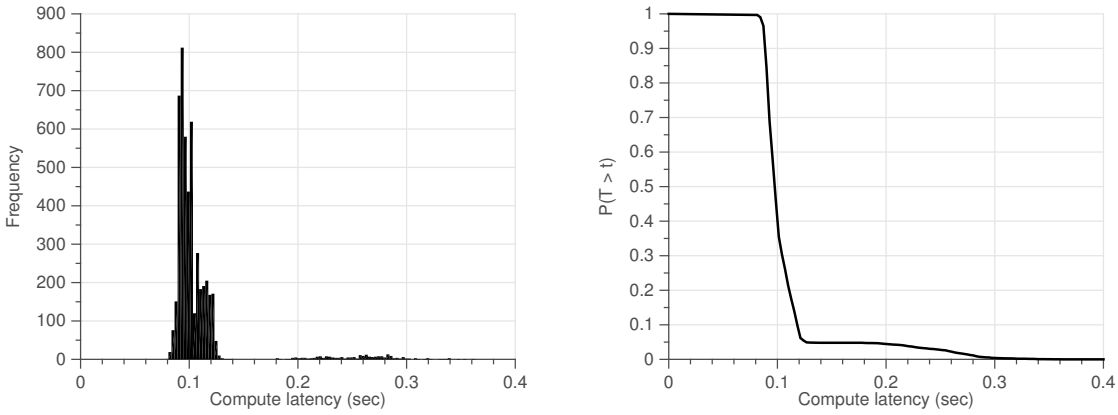
4 Coded Computation: Experiment Results

In order to see the efficacy of coded computation, we implement the proposed algorithms and test them on an Amazon EC2 cluster. In this section, we provide the experiment setups and the results.

4.1 Task runtime

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. 9, we plot the histogram and CCDF of measured computing times; the average round trip time is 0.11 second, and the 95^{th} percentile latency is 0.20 second, i.e., roughly five out of hundred tasks are going to be roughly



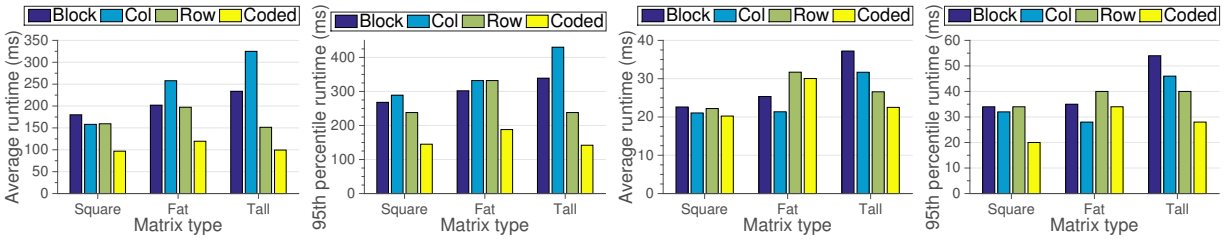
(a) Histogram of the measured round trip times

(b) Empirical CCDF of the measured round trip times.

Figure 9: **Histogram and 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.

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. 6a, we plot the runtime distributions of uncoded/coded distributed algorithms using this empirical distribution as the 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.

4.2 Coded Matrix Multiplication



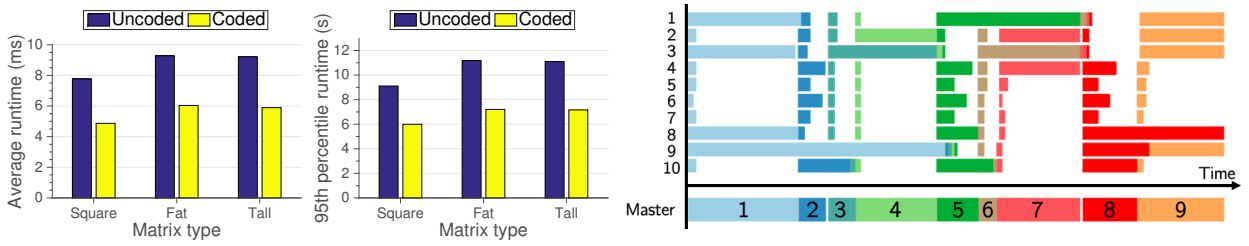
(a) `m1-small`, average runtime (b) `m1-small`, tail runtime (c) `c1-medium`, average runtime (d) `c1-medium`, tail runtime

Figure 10: **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^{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.

The coded matrix multiplication is implemented in C++ using OpenMPI [65] and benchmarked on a cluster of 26 EC2 instances (25 workers and a master). We manage the cluster using the StarCluster toolkit [66]. Input data is generated using a Python script, and the input matrix was 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 [67] 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. Similarly, 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. 10 shows that the coded matrix multiplication outperforms all the other parallel matrix multiplication algorithms in most cases. On a cluster of `m1-small` instances, compared to the best of the 3 uncoded matrix multiplication algorithms, the coded matrix multiplication achieves about 40% average runtime reduction and about 60% tail reduction. 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^{th} percentile runtime is reduced by at most 58.3%.

4.3 Coded Linear Regression



(a) Average (b) Tail (c) Gantt chart of coded gradient algorithm with an $(10, 8)$-MDS code

Figure 11: **Comparison of parallel gradient algorithms.** We compare parallel gradient algorithms for linear regression problems, with different matrix multiplication algorithms: row-partition and coded (row-partition) matrix multiplication. We implement the two gradient algorithms using Open MPI, and test them on an Amazon EC2 cluster of 10 worker instances. We measure the average and the 95^{th} percentile runtime of the algorithms. Plotted in (a) and (b) are the measured results. We visualize via a Gantt chart how the coded gradient algorithm handles the stragglers in (c). In each row, we visualize which iteration each worker is in, and we visualize the master node's progress in the bottom row. With a $(10, 8)$-MDS coded matrix multiplication, the master node can proceed to the next iteration whenever the first 8 workers complete their local computations, treating the 2 stragglers as erasures.

We evaluate the performance of the parallel gradient descent algorithms for linear regression. Since matrix multiplication is the core block of the parallel gradient descent for linear regression, the performance of the underlying parallel matrix multiplication algorithm significantly impacts the performance of the overall algorithm's performance.

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). Like the matrix multiplication procedure, we generate the input data through a Python script. However, since both the transposed and untransposed versions of the input matrix are used in the linear regression algorithm, we duplicate the data and row-partition both versions of the input matrix (with the required encoding). The procedure begins by having all worker nodes load in their respective row-partitioned submatrices (both transposed and untransposed versions). Since each pass of linear regression consists of two matrix-vector multiplications, we split each pass into two iterations – an iteration for each matrix-vector multiplication (transposed multiply $Ax^{(t)}$, untransposed multiply $A^T z^{(t)}$). Then, for every iteration of the procedure, the master node sends the appropriate input vector to each worker node in the cluster through an asynchronous send (MPI_Isend). The message channel through which the input vector is sent determines whether this particular multiplication operation should be a transposed multiplication or not. On the worker side, each worker continually listens for work by probing (Iprobe) for the input vector messages. Upon reception of the input vector, the worker computes the partial matrix multiplication through a call to the BLAS routine. When this completes, it sends the result back to the master node on the same channel from which the message was received. During this process, the master node waits for a sufficient number of partial results by polling for messages (IProbe). When enough partial results have been obtained, the master node decodes the result of the matrix vector multiplication. Then, the necessary computation to move to the next iteration is performed and the next iteration is begun. Similarly, the uncoded linear regression algorithm that is based on the uncoded row-partition matrix multiplication is implemented and benchmarked together with the coded linear regression.

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. 11 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%.

To thoroughly understand how the iterative algorithm with coded matrix multiplication efficiently handles stragglers, we measure the progress of each worker and the master node while running the gradient algorithm with coded matrix multiplication. The Gantt chart depicted in Fig. 11c visualizes the experiment results. We can observe that since the master node can complete an iteration as soon as it collects any $n - 2 = 8$ workers, the master node can quickly proceed to the next iteration even with the existence of stragglers. This flexibility allows the overall algorithm to constantly progress without getting detained by ubiquitous stragglers, which significantly reduces the overall runtime of the algorithm.

Remark 4 (Cancellation of stragglers). In the above experiments on the coded linear regression, we did *not* cancel (or kill) straggler workers. The overhead of cancellation of stragglers in our implementation is too high, so we choose not to cancel those stragglers. In an iterative algorithm such as the coded gradient descent, not canceling stragglers dilutes the gain of codes because only k workers will be available at the beginning of the following iteration. We believe that a delicate implementation of cancellation mechanism can even further improve the runtime performance of iterative coded distributed algorithms.

5 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. In Section. 6, we provide extensive simulation and experiment results that corroborate our theoretical findings.

5.1 Setup

We consider a master-worker distributed setup, where the master node has access to the entire data-set. Before each *epoch* of the distributed algorithm, the master node shuffles the data points and sends each worker node some *coded functions* of the data, the worker nodes use the received information to decode and extract actual (shuffled) data points and train a local model; at the end of an epoch, the local models are averaged, and the process is repeated. See Fig. 12 for a toy illustration. We design a transmission strategy for the worker node, and caching and decoding strategies for

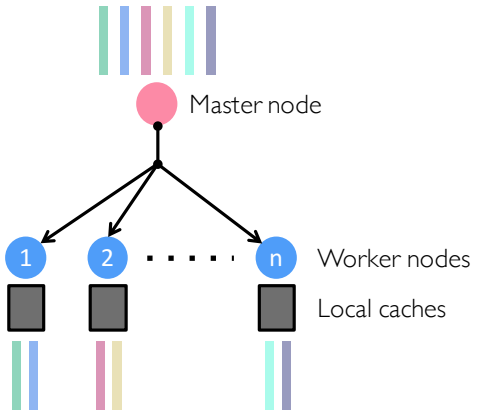


Figure 12: **Distributed setup.** We consider a master worker setup, where the master node communicates data points (or coded functions of them), and the worker nodes can store a limited number of data points. The topology of the network (be it tree-like, mesh, or over a shared bus) allows for a degree of broadcasting gains. That is, we assume that broadcasting the same information to all users is "cheaper" than sending them individual messages.

the worker nodes that minimize the data communicated across all the shufflings performed. For completeness, we present the high-level description of a distributed machine learning proto-algorithm. Let $A \in \mathbb{R}^{q \times r}$ be a data matrix. Consider an optimization problem that can be expressed as $\min_{x \in \mathbb{R}^r} \; f(x) = \sum_{i=1}^{q} \ell_i(a_i, x)$, where a_i is the i-th data row of A, and ℓ_i is a local function of the variable x and data point a_i. The above problem can be solved by an iterative distributed algorithm that operates on rounds, where at each round each worker locally trains a model (variable) x_i of dimension r that is communicated back to the master. Upon receiving all the local models, the master averages them into a single model, and broadcasts it back to the workers. More precisely, at iteration t of the algorithm, the data set is partitioned randomly into n subsets, say $A_1, A_2, \ldots, A_n$. Worker i computes a local update vector $x_i = h(x^t, A_n)$, where x^t is the model (variable) at iteration t. The master node then aggregates the results by simply averaging

the local updates. The algorithm continues by iterating $x^{t+1} = \frac{1}{n}\sum_{k=1}^{n} h(x^t, A_k)$. A prototypical exemplar of the described parallel learning proto-algorithm is the parallel stochastic gradient descent [68].

5.2 Coded Shuffling Algorithm

Before we rigorously explain the technical details of the coded shuffling algorithm and our preliminary results, we provide a toy example (shown in Fig. 13) that illustrates how codes can reduce the cost of the communication phase during the execution of distributed learning algorithms.

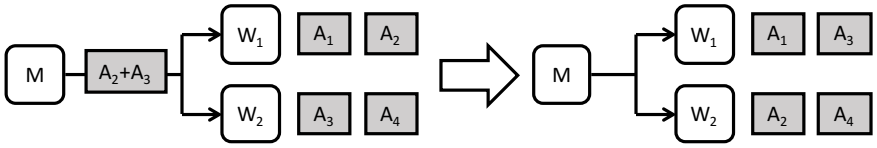


Figure 13: **Illustration of *Coded Shuffling*.** The data matrix $A \in \mathbb{R}^{q\times r}$ is partitioned into 4 submatrices: $A_1 \in \mathbb{R}^{q/4\times r}$ to $A_4 \in \mathbb{R}^{q/4\times r}$. Before shuffling, worker 1 has cached A_1 and A_2 (from a previous iteration of the algorithm), and worker 2 has cached A_3 and A_4. Assuming that a new shuffling requires node 1 to receive A_3 and node 2 to receive A_2, the master node can send $A_2 + A_3$ (the addition is over the representation field of A, not over bits) in order to shuffle the data stored at the two workers. Observe that in this case, and by using the cached information, the amount of communication (assuming full broadcast gain over uni-casting) is 50%.

Consider a system with two worker nodes and one master node. Assume that the master node holds the entire data set A. For clarity of exposition of this example, assume that the data is equipartitioned into 4 batches $A_1, \ldots, A_4$. Assume that worker 1 already has A_1 and A_2 cached locally, and worker 2 has A_3 and A_4 cached locally. To shuffle the data, the master node's objective is to transmit A_3 to worker 1 and A_4 to worker 2, so that the local functions are now computed on the data points of (A_1, A_3) and (A_2, A_4) by worker nodes 1 and 2, respectively. For this purpose, the master node can simply broadcast a *coded* message $A_2 + A_3$ to the worker nodes. Since node 1 has access to A_2, it can subtract A_2 from the received message $A_2 + A_3$, and replace A_2 with A_3. Similarly, node 2 can replace A_3 with A_2. Compared to the naïve (or uncoded) shuffling scheme in which the master node transmits A_2 and A_3 separately, this new shuffling scheme can save 50% of the communication cost, speeding up the overall run-time of the distributed machine learning algorithm. This is true assuming that *broadcasting* a message to all workers is significantly cheaper than sending individual messages to each worker; that is, assuming that $\gamma(n) = n$. Recall that $\gamma(n)$ is the ratio of the cost of unicasting n messages to n users to broadcasting 1 message (of the same size) to n users. This assumption is clearly the case for wireless architecture that is of great interest with the emerge of wireless data centers, e.g. [69, 70], and mobile computing platforms [71]. Interestingly, we observe that $\gamma(n) \gg 1$ even for simple networked (i.e., non-wireless) distributed systems. For instance, the Message Passing Interface (MPI) broadcast (`MPI_Bcast`) utilizes a tree broadcast algorithm, which achieves $\gamma(n) = \Theta(\frac{n}{\log(n)})$. We experimentally demonstrate the gain of broadcast over unicast in open MPI. We measure the time taken for a data block to be transmitted to a targeted number of workers on an Amazon EC2 cluster, and compare the average transmission time taken with MPI unicast and that with MPI broadcast. Fig. 14 illustrates that using unicast, the average transmission time increases linearly as the number of receivers increases as expected, but with MPI broadcast, the average transmission time increases only logarithmically. (See Remark 5 for more detailed discussion.)

General Coded Shuffling Scheme To formally describe the coded shuffling algorithm, we define some notation. Let $A(\mathcal{J}) \in \mathbb{R}^{|\mathcal{J}|\times r},\ \mathcal{J} \subset [q]$ be the concatenation of $|\mathcal{J}|$ rows of matrix 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, the set of data rows $[q] \triangleq \{1, 2, \ldots, q\}$ is partitioned uniformly at random into n subsets $S_i,\ 1 \leq i \leq n$ so that $\cup_{i=1}^{n} S_i = [q]$ and $S_i \cap S_j = \emptyset$ when $i \neq j$; thus, each worker node computes a fresh local function of the data. Clearly, the data set that worker i works on has cardinality q/n, i.e., $|S_i| = q/n$. We refer to each of these subsets as a *mini-batch* of the dataset. Since, the mini-batches are non-overlapping, the data sampling we consider here is *without replacement*.

We now provide details of the coded shuffling algorithm after each iteration of the parallel machine learning algorithm. 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

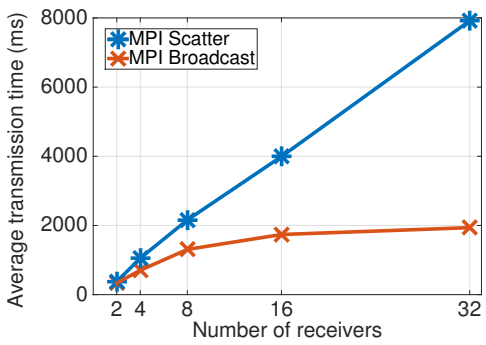


Figure 14: **Gains of broadcasting over uni-casting in distributed systems.** We measure the time taken for a data block of size of 4.15 MB to be transmitted to a targeted number of workers on an Amazon EC2 cluster, and compare the average transmission time taken with Message Passing Interface (MPI) 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.

(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.

After iteration t, the master node aggregates the local functions by averaging them, draws a random permutation on $[q]$, say π^t, to find S_i^{t+1}, and transmits a message $m(t+1)$ (i.e., a number of "coded" data-points) such that S_i^{t+1} can be recovered by worker node i from the current cache content C_i^t and the transmitted message $m(t+1)$. The cache content is then updated according to the following rule: the new cache will contain the subset of the data points used in the current iteration (this is need for the local computation), plus a random subset of the previous cached contents. 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}, $1 \leq i \leq n$ 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.

Example 3. To further clarify the algorithm, we revisit our toy example again based on this notation. Let $n = 2$ and $s = q/2$. Each worker node has half of the data stored in its cache. Consider an iteration of the algorithm, where a new permutation of the data rows is drawn by the master node resulting in subsets S_1 and S_2. As q gets large, half of the mini-batch S_1 (that should be processed by worker node 1) is stored in cache 1, i.e. $|S_1 \cap C_1| = q/4$. Similarly, $|S_2 \cap C_2| = q/4$. Thus, without the ability to exploit the power of coding, the master node needs to transmit a total of $q/4 + q/4 = q/2$ data rows to the computing nodes that are rows corresponding to $S_1 \cap \bar{C}_1$ and $S_2 \cap \bar{C}_2$. By contrast, in order to exploit the power of coding, the key observation is that the data rows that are in S_1 but not stored in C_1, are stored in C_2 (since $S_1 \cup S_2 = [q]$), and vice versa. Thus, the master node can code by sending only the *sum* of the rows indexed by $S_1 \cap C_2$ and $S_2 \cap C_1$, denoted by $A(S_1 \cap C_2) + A(S_2 \cap C_1)$. The transmission rate is thus $q/4$, which leads to a factor of 2 reduced communication cost. For decoding, since computing node 1 has access to $A(S_2 \cap C_1)$, it can recover $A(S_1 \cap C_2)$. Similarly, node 2 can recover $A(S_2 \cap C_1)$.

We now describe two methods for transmitting the message $m(t)$: (1) uncoded transmission and (2) coded transmission.

Uncoded Transmission Consider the cache content C_i^t, $1 \leq i \leq n$, and the new data rows required by each worker, S_i^{t+1}, $1 \leq i \leq n$. In the following description, without loss of generality, we drop the iteration index t (and $t+1$) for the ease of notation. We find how many data rows in S_i are already cached in C_i, i.e. we find $|C_i \cap S_i|$. 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 $|C_i \cap S_i| = \frac{q}{n}(1 - s/q)$. 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

$$R_u = n \times \frac{q}{n}(1 - s/q) = q(1 - s/q). \tag{16}$$

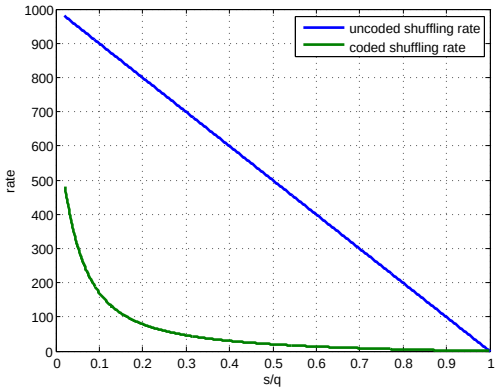


Figure 15: **The achievable rates of coded and uncoded shuffling schemes.** This figure shows the achievable rates of coded and uncoded schemes versus the cache size for parallel stochastic gradient descent algorithm.

Coded Transmission The delivery algorithm of the coded transmission is as follows. Define the set of "exclusive" cache content as $\tilde{C}_{\mathcal{I}} = \cap_{i\in\mathcal{I}} C_i \cap_{i'\in[n]\setminus\mathcal{I}} \bar{C}_{i'}$ that denotes the set of rows that are stored at the caches of $\mathcal{I}$, and are *not* stored at the caches of $[q] \setminus \mathcal{I}$. For each subset $\mathcal{I}$ with $|\mathcal{I}| \geq 2$, the master node will broadcast $\sum_{k\in\mathcal{I}} A(S_k \cap \tilde{C}_{\mathcal{I}\setminus\{k\}})$ to the worker nodes. This is the summation (over k) of the data points that should be processed by worker k, and these data points are not stored in the cache of worker k, but are instead stored in the caches of every other worker in set $\mathcal{I}$. Thus, this message enables simultaneous decoding of some missing data points for all the workers in $\mathcal{I}$, by exploiting this multicast coding opportunity. This completes the description of the coded transmission algorithm. A key novelty of our scheme is that the coding is performed over the representation field of the data matrix A, and not over bits. The following example further illustrates the explained procedure.

Example 4. Let $n = 3$. Note that worker node k needs to find $S_k \cap \bar{C}_k$ upon receiving the transmitted message from the master node. Consider $k = 1$. The data rows in $S_1 \cap \bar{C}_1$ are stored either exclusively in C_2 or C_3 (i.e. $\tilde{C}_2$ or $\tilde{C}_3$), or stored in both C_2 and C_3 (i.e. $\tilde{C}_{2,3}$). The transmitted message consists of 4 parts: 1) the part corresponding to $\mathcal{I} = \{1, 2\}$ is $M_1 = A(S_1 \cap \tilde{C}_2) + A(S_2 \cap \tilde{C}_1)$, 2) the part corresponding to $\mathcal{I} = \{1, 3\}$ is $M_2 = A(S_1 \cap \tilde{C}_3) + A(S_3 \cap \tilde{C}_1)$, 3) the part corresponding to $\mathcal{I} = \{2, 3\}$ is $M_3 = A(S_2 \cap \tilde{C}_3) + A(S_3 \cap \tilde{C}_2)$, and 4) the part corresponding to $\mathcal{I} = \{1, 2, 3\}$ is $M_4 = A(S_1 \cap \tilde{C}_{2,3}) + A(S_2 \cap \tilde{C}_{1,3}) + A(S_3 \cap \tilde{C}_{1,2})$. We show that worker node 1 can recover its unstored data rows, $S_1 \cap \bar{C}_1$. From part 1 of the message, it recovers $A(S_1 \cap \tilde{C}_2) = M_1 - A(S_2 \cap \tilde{C}_1)$. Observe that node 1 has access to $S_2 \cap \tilde{C}_1$.[4] From part 2 of the message, node 1 recovers $A(S_1 \cap \tilde{C}_3) = M_2 - A(S_3 \cap \tilde{C}_1)$. Finally, from part 4 of the message, node 1 recovers $A(S_1 \cap \tilde{C}_{2,3}) = M_4 - A(S_2 \cap \tilde{C}_{1,3}) - A(S_3 \cap \tilde{C}_{1,2})$.

5.3 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 5 (Coded Shuffling Rate). *Coded shuffling achieves communication rate*

$$R_c = \frac{q}{(np)^2}\left((1-p)^{n+1} + (n-1)p(1-p) - (1-p)^2\right) \tag{17}$$

(in number of data rows transmitted per iteration from the master node), which is significantly smaller than R_u in (16)*.*

The reduction in communication rate is illustrated in Fig. 15 for $n = 50$ and $q = 1000$ as a function of s/q, where $1/n \leq s/q \leq 1$. 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. We now prove Theorem 5.

[4]The master node also transmits the new permutation that has negligible overhead compared to the data that needs to be transmitted.

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

$$\Pr(k \in \tilde{C}_{\mathcal{I}}^t) = \sum_{i=1}^{n} \Pr(k \in \tilde{C}_{\mathcal{I}}^t | k \in S_i^t) \Pr(k \in S_i^t) \tag{18}$$

$$= \sum_{i=1}^{n} \frac{1}{n} \Pr(k \in \tilde{C}_{\mathcal{I}}^t | k \in S_i^t) \tag{19}$$

$$= \sum_{i \in \mathcal{I}} \frac{1}{n} \Pr(k \in \tilde{C}_{\mathcal{I}}^t | k \in S_i^t) \tag{20}$$

$$= \sum_{i \in \mathcal{I}} \frac{1}{n} \left(\frac{s - q/n}{q - q/n} \right)^{|\mathcal{I}|-1} \left(\frac{q - s}{q - q/n} \right)^{n-|\mathcal{I}|} \tag{21}$$

$$= \frac{|\mathcal{I}|}{n} p^{|\mathcal{I}|-1} (1-p)^{n-|\mathcal{I}|}. \tag{22}$$

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

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

$$|S_k^{t+1} \cap \tilde{C}_{\mathcal{I}}^t| \simeq \frac{q}{n} \times \frac{|\mathcal{I}|}{n} p^{|\mathcal{I}|-1} (1-p)^{n-|\mathcal{I}|}.$$

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

$$R_c = \sum_{i=2}^{n} \binom{n}{i} \frac{q}{n} \frac{i-1}{n} p^{i-2} (1-p)^{n-(i-1)}. \tag{23}$$

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

$$\sum_{i=2}^{n} \binom{n}{i} (i-1) x^{i-2} = \frac{1 + (1+x)^{n-1}(nx - x - 1)}{x^2}. \tag{24}$$

Using (24) in (23) completes the proof. □

Corollary 6. *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 (17), after some manipulations, one calculates

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

which shows Corollary 6.

Corollary 7. *Consider the regime of interest where n, s, and q get large, and $s/q \to c > 0$ and $n/q \to 0$. Then,*

$$R_c \to q(1 - s/q) \frac{1}{ns/q} = \frac{R_u}{ns/q} \tag{25}$$

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

Remark 5. In many applications, the network topology is based on unicasting, and the multicasting opportunity is not fully available, or $\gamma(n) < n$. Then, the communication cost of uncoded shuffling is $R_u = n \times q/n(1 - s/q) = q(1 - s/q)$, and the communication cost of coded shuffling is

$$\frac{n}{\gamma(n)} R_c \simeq \frac{n}{\gamma(n)} q(1 - \frac{s}{q}) \frac{1}{ns/q} = \frac{R_u}{\gamma(n)s/q}.$$

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.

As previously explained , in many practical systems it is observed that *unicasting the same message to multiple nodes is significantly faster than unicasting different message (of the same size) to multiple nodes.*, justifying the advantage of using coded shuffling. For instance, the MPI broadcast API (`MPI_Bcast`) utilizes a tree broadcast algorithm, which achieves $\gamma(n) = \Theta(n/\log(n))$. Shown in Fig. 14 is the time taken for a data block to be transmitted to a various number of workers on an Amazon EC2 cluster. 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.

6 Coded Shuffling: Simulation Results

In this section, we simulate the performance of *Coded Shuffling*. More specifically, we compare the performance of parallel stochastic gradient descent (PSGD) algorithms with different shuffling schemes: *Coded Shuffling*, *Uncoded Shuffling*, and *No Shuffling*.

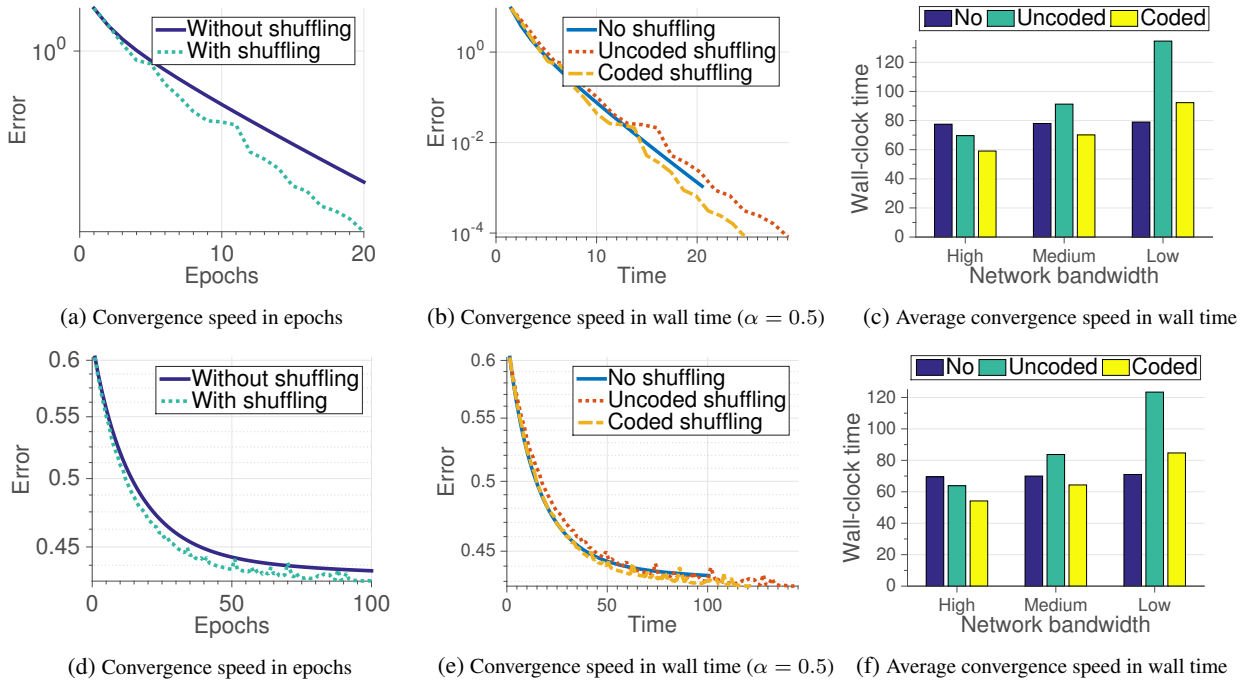
6.1 Linear Regression

We first simulate the performance of PSGD algorithm for a simple linear regression. That is, given a data matrix $A \in \mathbb{R}^{n \times d}$ and a vector $y \in \mathbb{R}^n$, we want to solve the following optimization problem: $\min_x ||Ax - y||_2^2$. We test the algorithm with both synthetic and real data. For the synthetic data, each element of the data matrix $A \in \mathbb{R}^{10^4 \times 10^3}$, $x^\star \in \mathbb{R}^{10^3}$, $w \in \mathbb{R}^{10^4}$ is chosen uniformly at random from $[0, 1]$. Then, $y = Ax^\star + w$ is given to the optimization problem as an input. With regard to experimental validation using "real" data, we use the Wine Quality Data Set from UCI Machine Learning Repository [72, 73]: Each row of the data matrix includes 11 physicochemical features of a wine such as pH, alcohol, density, acidity, etc., and the task is to predict the quality of a wine based on these features.

We plot in Fig. 16 the simulation results. The figures in the upper row are the simulation results with the synthetic data, and those in the lower row are the simulation results with the real data. We first compare the convergence performance of the PSGD with and without shuffling in Fig. 16a. We can see that the convergence rate is significantly improved by shuffling data between epochs. For instance, after 20 epochs, the algorithm with shuffling achieves a 10 times lower error performance compared to the one without shuffling. Let us now consider the cost of communication to be fair in our comparison of the schemes. In order to account for the cost of communication, we consider the *wall time*. The wall time consists of computation time and communication time: computation of each iteration is assumed to take a unit time, and communication of $\frac{1}{n}^{\text{th}}$ of the data matrix A is assumed to take α time units. Assuming n, s, and q are large and hence using Corollary 7, we can find the wall time after the i^{th} epoch is completed. We denote by $t_{\text{NS}}(i)$ the wall time after the i^{th} epoch when shuffling is not used, by $t_{\text{NS}}(i)$ the wall time when uncoded shuffling is used, and by $t_{\text{CS}}(i)$ the wall time when coded shuffling is used. Then,

$$t_{\text{NS}}(i) = \frac{\alpha}{n} + i, \quad t_{\text{US}}(i) = \frac{\alpha}{n} + \frac{i-1}{n}\left(1 - \frac{s}{q}\right) + i, \quad t_{\text{CS}}(i) = \frac{\alpha}{n} + \frac{i-1}{n}\left(1 - \frac{s}{q}\right)\frac{1}{1 + ns/q} + i.$$

Using the above equations, the convergence performance as a function of the wall time can be found. In Fig. 16, we compare the convergence performance of the three different schemes. Plotted in the upper row are the simulation results with synthetic data, and in the lower row are the ones with real data. In Fig. 16a and 16d, the convergence of the algorithm is shown without considering the cost of communication; the algorithm that shuffles data after each iteration converges significantly faster than the one that does not shuffle data. Plotted in Fig. 16b and 16e are the convergence of the algorithm measured in wall time. Although the algorithm with uncoded shuffling converges faster than the one without shuffling in terms of the number of epochs, when the communication cost is taken into account, the actual convergence time, measured in wall time, can be slower; however, the algorithm with coded shuffling still



(a) Convergence speed in epochs (b) Convergence speed in wall time ($\alpha = 0.5$) (c) Average convergence speed in wall time

(d) Convergence speed in epochs (e) Convergence speed in wall time ($\alpha = 0.5$) (f) Average convergence speed in wall time

Figure 16: **Linear regression**: Simulation results with synthetic (upper row) and real data (lower row). Plotted in (a) and (d) are the convergence performance of the algorithm with three different shuffling schemes: coded shuffling, uncoded shuffling, and no shuffling. In these figures, the x-axis is the number of passes, and the y-axis is the error. Plotted in (b) and (e) are the convergence performance of the algorithm as a function of the wall time. Plotted in (c) and (f) are the average convergence performance in wall time: we compute the wall clock time to achieve a target error with different values of bandwidth – high network bandwidth ($\alpha = 0.5$), medium network bandwidth ($\alpha = 1$), and low network bandwidth ($\alpha = 2$) –, and find the average convergence time in wall clock time over 100 runs.

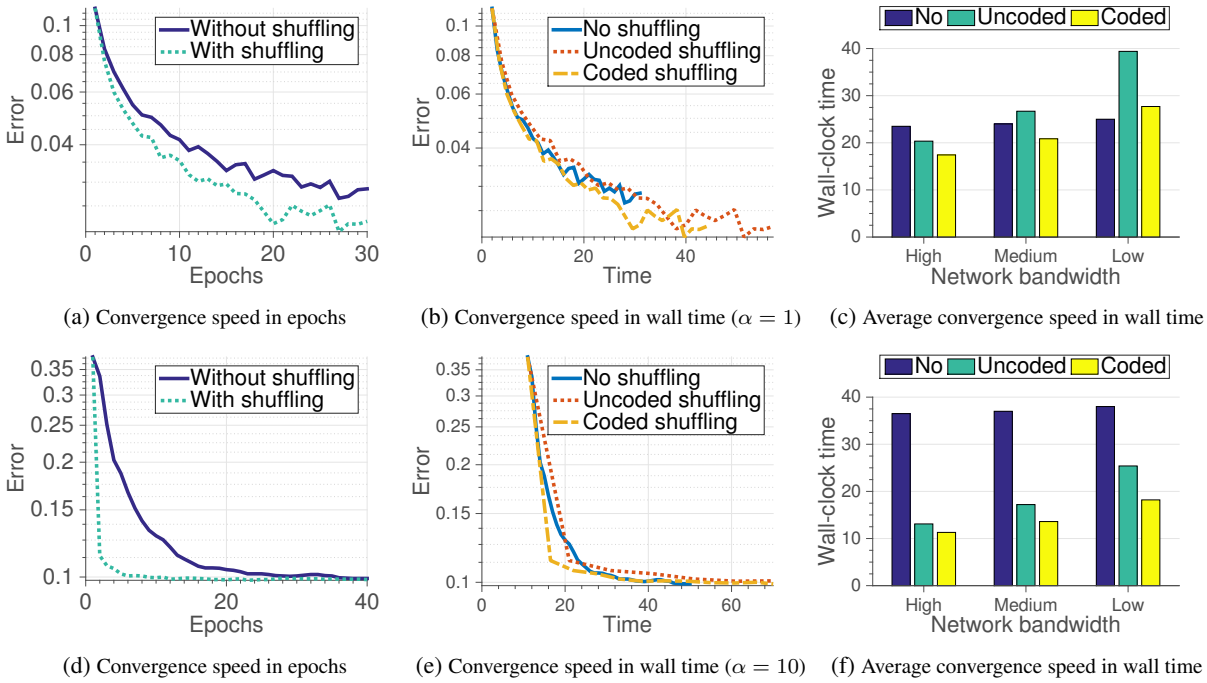
converges faster than the others because of its reduced communication overhead. We vary the network bandwidth (communication overhead α) and measure the average convergence time (in wall time) over 100 runs, and the results are shown in Fig. 16c and 16f. The algorithm with coded shuffling converges faster than the the others in all the cases, except for the cases where the bandwidth is too low (communication cost is too high).

6.2 Classification

We also compare the performance of the algorithms for classification. More specifically, we solve a logistic regression problem and classify all the data points into two classes; given a data matrix $A = (a_1, a_2, \ldots, a_q)^T \in \mathbb{R}^{q \times r}$ and a label vector $y \in \{0, 1\}^q$, we want to solve the following optimization problem: $\min_x \sum_{i=1}^{q} -\log \Pr(y_i | a_i; x)$, where $\Pr(y_i = 0 | a_i; x) = \frac{\exp(-a_i^T x)}{1+\exp(-a_i^T x)}$ and $\Pr(y_i = 1 | a_i; x) = \frac{1}{1+\exp(-a_i^T x)}$. We run the PSGD algorithm to solve the above optimization problem and measure the convergence performance of the algorithm with the different shuffling schemes.

For synthetic data, each element of the data matrix $A \in \mathbb{R}^{10^4 \times 10^3}$ is drawn from standard normal distribution, and each element of $x^\star \in \mathbb{R}^{10^3}$ is chosen uniformly at random from $[0, 1]$. Then, y is drawn according to the logistic distribution and is given to the optimization problem as an input. For the real data, we use Spambase Data Set from UCI Machine Learning Repository [73]. Each row of the data matrix includes 48 attributes of emails, and the task is to determine whether an email is spam or not based on these features.

Plotted in Fig. 17 are classification simulation results with synthetic and real data. Similar to the simulation results with linear regression, we observe that shuffling significantly improves the convergence performance of the PSGD algorithms, as shown in Fig. 17a and 17d. When communication overhead is considered, the algorithm with coded shuffling provides the best performance among the three schemes unless the communication cost is too high.



(a) Convergence speed in epochs (b) Convergence speed in wall time ($\alpha = 1$) (c) Average convergence speed in wall time

(d) Convergence speed in epochs (e) Convergence speed in wall time ($\alpha = 10$) (f) Average convergence speed in wall time

Figure 17: **Classification**: Simulation results with synthetic (upper row) and real data (lower row). Plotted in (a) and (d) are the convergence performance of the algorithm with three different shuffling schemes: coded shuffling, uncoded shuffling, and no shuffling. In these figures, the x-axis is the number of passes, and the y-axis is the error. Plotted in (b) and (e) are the convergence performance of the algorithm as a function of the wall time. Plotted in (c) and (f) are the average convergence performance in wall time: we compute the wall clock time to achieve a target error with different values of bandwidth – high network bandwidth ($\alpha = 0.5$), medium network bandwidth ($\alpha = 1$), and low network bandwidth ($\alpha = 2$) –, and find the average convergence time in wall clock time over 100 runs.

7 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 cleverly introducing redundancy through codes into the computation. Our novel *Coded Shuffling* framework can significantly reduce the overhead of data-shuffling, which is a costly process used in current machine-learning algorithms and is required for achieving high statistical efficiency. Our preliminary experiment results validate the power of our proposed schemes in effectively curtailing the negative effects of system bottlenecks, and attaining a 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. We believe that for a broader class of functions there exists a fundamental tradeoff between system resources – the computation load of each node, the runtime of distributed algorithm, the storage overheads, and the number of workers –, and we are actively studying this new tradeoff.

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 $\mathcal{O}(\frac{1}{n})$, 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] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. of the 6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.

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

[4] 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 Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, 2011.

[5] C. Suh and K. Ramchandran, "Exact-repair MDS code construction using interference alignment," *IEEE Transactions on Information Theory*, vol. 57, no. 3, pp. 1425–1442, 2011.

[6] I. Tamo, Z. Wang, and J. Bruck, "MDS array codes with optimal rebuilding," in *Proc. of IEEE International Symposium on Information Theory (ISIT)*, 2011.

[7] V. R. Cadambe, C. Huang, S. A. Jafar, and J. Li, "Optimal repair of MDS codes in distributed storage via subspace interference alignment," *arXiv preprint arXiv:1106.1250*, 2011.

[8] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe, "Repair optimal erasure codes through Hadamard designs," in *Proc. of the 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2011.

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

[10] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *Proc. of IEEE INFOCOM*, 2011.

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

[12] J. Han and L. A. Lastras-Montano, "Reliable memories with subline accesses," in *Proc. of IEEE International Symposium on Information Theory (ISIT)*, 2007.

[13] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," in *Proc. of the Sixth IEEE International Symposium on Network Computing and Applications (NCA)*, 2007.

[14] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," in *Proc. of IEEE International Symposium on Information Theory (ISIT)*, 2012.

[15] G. M. Kamath, N. Prakash, V. Lalitha, and P. V. Kumar, "Codes with local regeneration," *arXiv preprint arXiv:1211.1932*, 2012.

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

[17] N. Prakash, G. M. Kamath, V. Lalitha, and P. V. Kumar, "Optimal linear codes with a local-error-correction property," in *Proc. of IEEE International Symposium on Information Theory (ISIT)*, 2012.

[18] N. Silberstein, A. Singh Rawat, and S. Vishwanath, "Error resilience in distributed storage via rank-metric codes," *CoRR*, vol. abs/1202.0800, 2012.

[19] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in Windows Azure Storage," in *Proc. of USENIX Annual Technical Conference (ATC)*, Jun. 2012.

[20] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: Novel erasure codes for big data," in *Proc. of the VLDB Endowment*, vol. 6, no. 5, 2013.

[21] 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. of USENIX HotStorage*, Jun. 2013.

[22] 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. of the ACM conference on SIGCOMM*, 2014.

[23] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[24] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in Map-Reduce clusters using Mantri," in *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[26] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Proc. of the 25th Annual Conference on Neural Information Processing Systems (NIPS)*, 2011.

[27] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. of the 25th Annual Conference on Neural Information Processing (NIPS)*, 2011.

[28] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[29] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" in *Proc. of the 51st Annual Allerton Conference on Communication, Control, and Computing*, 2013.

[30] D. Wang, G. Joshi, and G. W. Wornell, "Efficient task replication for fast response times in parallel computation," in *Proc. of ACM SIGMETRICS*, 2014.

[31] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytiä, "Reducing latency via redundant requests: Exact analysis," in *Proc. of ACM SIGMETRICS*, 2015.

[32] M. Chaubey and E. Saule, "Replicated data placement for uncertain scheduling," in *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPS)*, 2015.

[33] K. Lee, R. Pedarsani, and K. Ramchandran, "On scheduling redundant requests with cancellation overheads," in *Proc. of the 53rd Annual Allerton conference on Communication, Control, and Computing*, Oct. 2015.

[34] D. P. Bertsekas, *Nonlinear programming*. Athena scientific, 1999.

[35] A. Nedic and A. E. Ozdaglar, "Distributed subgradient methods for multi-agent optimization," *IEEE Transactions on Automatic Control*, vol. 54, no. 1, pp. 48–61, 2009.

[36] S. P. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[37] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

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

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

[40] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proc. of the 26th Annual Conference on Neural Information Processing Systems (NIPS)*, 2012.

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

[42] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "MLbase: A distributed machine-learning system," in *Proc. of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[43] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: an API for distributed machine learning," in *Proc. of the IEEE 13th International Conference on Data Mining (ICDM)*, 2013.

[44] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[45] B. Recht and C. Ré, "Parallel stochastic gradient algorithms for large-scale matrix completion," *Mathematical Programming Computation*, vol. 5, no. 2, pp. 201–226, 2013.

[46] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade - Second Edition*, 2012, pp. 421–436.

[47] C. Zhang and C. Re, "Dimmwitted: A study of main-memory statistical analytics," *PVLDB*, vol. 7, no. 12, pp. 1283–1294, 2014.

[48] M. Gürbüzbalaban, A. Ozdaglar, and P. Parrilo, "Why random reshuffling beats stochastic gradient descent," *arXiv preprint arXiv:1510.08560*, 2015.

[49] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[50] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," *IEEE Transactions on Information Theory*, vol. 60, no. 5, pp. 2856–2867, 2014.

[51] M. A. Maddah-Ali and U. Niesen, "Decentralized coded caching attains order-optimal memory-rate tradeoff," *IEEE/ACM Transactions on Networking*, vol. 23, no. 4, pp. 1029–1040, 2015.

[52] R. Pedarsani, M. A. Maddah-Ali, and U. Niesen, "Online coded caching," in *Proc. of IEEE International Conference on Communications (ICC)*, 2014.

[53] N. Karamchandani, U. Niesen, M. A. Maddah-Ali, and S. Diggavi, "Hierarchical coded caching," in *Proc. of IEEE International Symposium on Information Theory (ISIT)*, 2014.

[54] M. Ji, G. Caire, and A. F. Molisch, "Fundamental limits of distributed caching in D2D wireless networks," in *Proc. of IEEE Information Theory Workshop (ITW)*, 2013.

[55] S. Li, M. A. Maddah-ali, and S. Avestimehr, "Coded MapReduce," Presented at the 53rd Annual Allerton conference on Communication, Control, and Computing, Monticello, IL, 2015.

[56] J. A. Bilmes, K. Asanovic, C. Chin, and J. Demmel, "Author retrospective for optimizing matrix multiply using PHiPAC: a portable high-performance ANSI C coding methodology," in *Proc. of ACM International Conference on Supercomputing 25th Anniversary*, 2014.

[57] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel strassen: implementation and performance," in *Proc. of High Performance Computing Networking, Storage and Analysis (SC)*, 2012.

[58] J. Demmel, "Communication–avoiding algorithms for linear algebra and beyond," Presented at IEEE 27th International Symposium on Parallel Distributed Processing, 2013.

[59] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Scientific Computing*, vol. 34, no. 1, 2012.

[60] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.

[61] D. Costello and S. Lin, *Error control coding*. New Jersey, 2004.

[62] G. Liang and U. C. Kozat, "TOFEC: achieving optimal throughput-delay trade-off of cloud storage using erasure codes," in *Proc. of IEEE Conference on Computer Communications (INFOCOM)*, 2014.

[63] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[64] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in apache spark," *CoRR*, vol. abs/1505.06807, 2015.

[65] "Open MPI: Open source high performance computing," http://www.open-mpi.org, accessed: 2015-11-25.

[66] "StarCluster," http://star.mit.edu/cluster/, accessed: 2015-11-25.

[67] "BLAS (Basic Linear Algebra Subprograms)," http://www.netlib.org/blas/, accessed: 2015-11-25.

[68] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized stochastic gradient descent," in *Proc. of the 24th Annual Conference on Neural Information Processing (NIPS)*, 2010.

[69] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting data center networks with multi-gigabit wireless links," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, 2011.

[70] Y. Zhu, X. Zhou, Z. Zhang, L. Zhou, A. Vahdat, B. Y. Zhao, and H. Zheng, "Cutting the cord: a robust wireless facilities network for data centers," in *Proc. of the 20th annual international conference on Mobile computing and networking*, 2014.

[71] 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. of the 8th international conference on Emerging networking experiments and technologies*, 2012.

[72] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Modeling wine preferences by data mining from physicochemical properties," *Decision Support Systems*, vol. 47, no. 4, pp. 547–553, 2009.

[73] M. Lichman, "UCI machine learning repository," 2013, accessed: 2015-08-01. [Online]. Available: http://archive.ics.uci.edu/ml