Coded Computing®

Mitigating Fundamental Bottlenecks in Large-scale Distributed Computing and Machine Learning

Suggested Citation: Songze Li and Salman Avestimehr (2016), "Coded Computing®", : Vol. xx, No. xx, pp 1–16. DOI: 10.1561/XXXXXXXXX.

Songze Li
University of Southern California
songzeli@usc.edu

Salman Avestimehr
University of Southern California
avestimehr@ee.usc.edu

now
the essence of knowledge
Boston — Delft

Contents

Coded Computing®

Songze Li[1] and Salman Avestimehr[2]

[1] *University of Southern California; songzeli@usc.edu*
[2] *University of Southern California; avestimehr@ee.usc.edu*

ABSTRACT

We introduce the concept of "coded computing", a novel computing paradigm that utilizes coding theory to effectively inject and leverage data/computation redundancy to mitigate several fundamental bottlenecks in large-scale distributed computing, namely communication bandwidth, straggler's (i.e., slow or failing nodes) delay, privacy and security bottlenecks. More specifically, for MapReduce based distributed computing structures, we propose the "Coded Distributed Computing" (CDC) scheme, which injects redundant computations across the network in a structured manner, such that in-network coding opportunities are enabled to substantially slash the communication load to shuffle the intermediate computation results. We prove that CDC achieves the optimal tradeoff between computation and communication, and demonstrate its impact on a wide range of distributed computing systems from cloud-based datacenters to mobile edge/fog computing platforms. Secondly, to alleviate the straggler effect that prolongs the executions of distributed machine learning algorithms, we utilize the ideas from error correcting codes to develop "Polynomial Codes" for computing general matrix algebra, and "Lagrange Coded Computing" (LCC) for computing arbitrary multivariate polynomials. The core idea of these proposed schemes is to apply coding to create redundant data/computation scattered across the network, such that completing the overall

Songze Li and Salman Avestimehr (2016), "Coded Computing®", : Vol. xx, No. xx, pp 1–16. DOI: 10.1561/XXXXXXXXX.

computation task only requires a subset of the network nodes returning their local computation results. We demonstrate the optimality of Polynomial Codes and LCC in minimizing the computation latency, by proving that they require the least number of nodes to return their results. Finally, we generalize the LCC scheme to further provide security and privacy guarantees to the system. In particular, we demonstrate that with slightly more coding overhead, LCC protects the system against the largest number of adversarial/malicious workers, and to provide the same level of data privacy against colluding workers, LCC requires injecting the least amount of randomness. To illustrate the impact of coded computing on real world applications and systems, we implement the proposed coding schemes on cloud-based distributed computing systems, and significantly improve the run-time performance of important benchmarks including distributed sorting and distributed training of regression models, over state of the arts.

1

Introduction

Recent years have witnessed a rapid growth of large-scale machine learning and big data analytics, facilitating the developments of data-intensive applications like voice/image recognition, real-time mapping services, autonomous driving, social networks, and augmented/virtual reality. These applications are supported by cloud infrastructures composed of large datacenters. Within a datacenter, a massive amount of users' data are stored distributedly on hundreds of thousands of low-end commodity servers, and any application of big data analytics has to be performed in a distributed manner within or across datacenters. This has motivated the fast development of scalable, interpretable, and fault-tolerant distributed computing frameworks (see, e.g., Dean and Ghemawat, 2004; Zaharia *et al.*, 2010; Recht *et al.*, 2011; Gemulla *et al.*, 2011; Zhuang *et al.*, 2013) that efficiently utilize the underlying hardware resources (e.g., CPUs and GPUs).

In this monograph, we focus on addressing the following three major performance bottlenecks for large-scale distributed machine learning/data analytics systems.

- *Communication bottleneck*: Excessive data shuffling between compute nodes.

- *Straggler bottleneck*: Delay of computation caused by slow or failing compute nodes, which are referred to as stragglers.
- *Security bottleneck*: Vulnerability to eavesdroppers and attackers.

To alleviate these bottlenecks, we take an unorthodox approach by employing ideas and techniques from coding theory, and propose the concept of "coded computing", whose core spirit is described as follows.

Exploiting coding theory to optimally **inject** and **leverage** data/task **redundancy** in distributed computing systems, creating coding opportunities to overcome communication, straggler, and security bottlenecks.

Guided by this core spirit, we propose and evaluate a rich class of coded distributed computing frameworks, for computation tasks ranging from general MapReduce primitives to fundamental polynomial algebra, and for computation systems ranging from conventional cloud-based datacenters to emerging (mobile) edge/fog computing systems. In the rest of this chapter, we describe our contributions on utilizing coded computing to mitigate the communication, straggler, and security bottlenecks, and discuss related works.

Before proceeding with the overview of coded computing, we would like to also point out an important remark. In order to enable redundant computations in coded computing, we need to also redundantly store the datasets over which the computations are done. This would impose a certain communication and storage cost to the system. However, in many applications this cost can be ignored due to the following two reasons. First, in many computation scenarios we are interested in *many* computations over the *same* dataset (e.g., database query, keyword search, loss calculation in machine learning, etc). In those cases the cost of encoding and redundantly storing the dataset in the network can be amortized over many computations. Second, in many scenarios, the encoding and storage of the dataset can happen at a different time than the desired computations. For example, one can use the off-peak network times to properly encode and store the dataset, so as to be ready for computations during the peak times.

Coding for bandwidth reduction

It is well known that communicating intermediate computation results (or data shuffling) is one of the major performance bottlenecks for various distributed computing applications, including self-join Ahmad *et al.*, 2012, TeraSort Guo *et al.*, 2013, and many machine learning algorithms Chowdhury *et al.*, 2011. For instance, in a Facebook's Hadoop cluster, it is observed that 33% of the overall job execution time is spent on data shuffling Chowdhury *et al.*, 2011. Also as is observed in Zhang *et al.*, 2013, 70% of the overall job execution time is spent on data shuffling when running a self-join job on Amazon EC2 clusters. This bottleneck is becoming worse for training deep neural networks with millions of model parameters (e.g., ResNet-50 He *et al.*, 2016), where partial gradients with millions of entries are computed at distributed computing nodes and passed across the network to update the model parameters Chilimbi *et al.*, 2014.

Many optimization methods have been proposed to alleviate the communication bottleneck in distributed computing systems. For example, from the algorithm perspective, when the function that reduces the final result is commutative and associative, it was proposed to pre-combine intermediate results before data shuffling, cutting off the amount of data movement Dean and Ghemawat, 2004; Rajaraman and Ullman, 2011. On the other hand, from the system perspective, optimal flow scheduling across network paths has been designed to accelerate the data shuffling process Greenberg *et al.*, 2009; Al-Fares *et al.*, 2010, and distributed cache memories were utilized to speed up the data transfer between consecutive computation stages Zhang *et al.*, 2009; Ekanayake *et al.*, 2010. Recently, motivated by the fact that training algorithms exhibit tolerance to precision loss of intermediate results, a family of lossy compression (or quantization) algorithms for distributed learning systems have been developed to compress the intermediate results (e.g., gradients), and then the compressed results are communicated to achieve a smaller bandwidth consumption (see, e.g., Seide *et al.*, 2014; Alistarh *et al.*, 2017; Wen *et al.*, 2017; Bernstein *et al.*, 2018).

The above mentioned approaches are designed for specific computations and network structures, and difficult to generalize to handle arbitrary computation tasks. To overcome these difficulties, we focus on a general MapReduce-type distributed computing model Dean and Ghemawat, 2004, and propose to utilize coding theory to slash the communication bottleneck in running MapReduce applications. In particular, in this computing model, each input file is mapped into multiple intermediate values, one for each of the output functions, and the intermediate values from all input files for each output function are collected and reduced to the final output result. For this model, we propose a coded computing scheme, named "coded distributed computing" (CDC), which trades extra local computations for more network bandwidth. For some design parameter r, which is termed as "communication load", the CDC scheme places and maps each of the input files on r carefully chosen distributed computing nodes, injecting r times more local computations. In return, the redundant computations produce side information at the nodes, which enable the opportunities to create *coded multicast* packets during data shuffling that are simultaneously useful for r nodes. That is, the CDC scheme trades r times more redundant computations for an r times reduction in the communication load. Furthermore, we theoretically demonstrate that this inversely proportional tradeoff between computation and communication achieved by CDC is fundamental, i.e., for a given computation load, no other schemes can achieve a lower communication load than that achieved by CDC.

Having proposed the CDC framework and characterized its optimal performance in trading extra computations for communication bandwidth, we also empirically demonstrate its impact on speeding up practical workloads. In particular, we integrate the principle of CDC into the widely used Hadoop sorting benchmark, `TeraSort` *Hadoop TeraSort* n.d., developing a novel distributed sorting algorithm, named `CodedTeraSort`. At a high level, `CodedTeraSort` imposes *structured* redundancy in the input data, enabling in-network coding opportunities to significantly slash the load of data shuffling, which is a major bottleneck of the run-time performance of `TeraSort`. Through extensive experiments on Amazon EC2 *Amazon Elastic Compute Cloud (EC2)* n.d. clusters, we demonstrate that `CodedTeraSort` achieves $1.97\times \sim 3.39\times$ speedup over `TeraSort`, for typical settings of interest. Despite the extra overhead imposed by coding (e.g., generation of the coding plan, data encoding and decoding), the practically achieved performance gain approximately matches the gain theoretically promised by `CodedTeraSort`.

Beyond the conventional wireline networks in datacenters, we also introduce the concept of coded computing to tackle the scenarios of mobile edge/fog computing, where the communication bottleneck is even more severe due to the low data rate and the large number of mobile users. In particular, we consider a wireless distributed computing platform, which is composed of a cluster of mobile users scattered around the network edge, connected wirelessly through an access point. Each user has a limited storage and processing capability, and the users have to collaborate to satisfy their computational needs that require processing a large dataset. This ad hoc computing model, in contrast to the centralized cloud computing model, is becoming increasingly common in the emerging edge computing paradigm for Internet-of-Things (IoT) applications Bonomi *et al.*, 2012; Chiang and Zhang, 2016. For this model, following the principle of the CDC scheme, we propose a coded wireless distributed computing (CWDC) scheme that jointly designs the local storage and computation for each user, and the communication schemes between the users. The CWDC scheme achieves a constant bandwidth consumption that is independent of the number of users in the network, which leads to a *scalable* design of the platform that can simultaneously accommodate an arbitrary number of users. Moreover, for a more practically important decentralized setting, in which each user needs to decide its local storage and computation independently without knowing the existence of any other participating users, we extend the CWDC scheme to achieve a bandwidth consumption that is very close to that of the centralized setting.

Coding for straggler mitigation

Other than data shuffling, another major performance bottleneck of distributed computing applications is the effect of stragglers. That is, the execution time of a computation consisting of multiple parallel tasks is limited by the slowest task run on the straggling processor. These stragglers significantly slow down the overall computations, and have been widely observed in distributed computing systems (see, e.g., Zaharia *et al.*, 2008; Ananthanarayanan *et al.*, 2013; Dean and Barroso, 2013). For instance, it was experimentally demonstrated in Zaharia *et al.*, 2008 that this straggler effect can prolong the job execution time by as much as 5 times.

Conventionally, in the original open-source implementation of Hadoop MapReduce *Apache Hadoop* n.d., the stragglers are constantly detected and the slow tasks are speculatively restarted on other available nodes. Following this idea of straggler detection, more timely straggler detection algorithms and better scheduling algorithms have been developed to further alleviate the straggler effect (see, e.g., Ananthanarayanan *et al.*, 2010; Zaharia *et al.*, 2008). Apart from straggler detection and speculative restart, another straggler mitigation technique is to schedule the clones of the same task (see, e.g., Ananthanarayanan *et al.*, 2013; Gardner *et al.*, 2015; Lee *et al.*, 2015; Chaubey and Saule, 2015; Shah *et al.*, 2016). The underlying idea of cloning is to execute redundant tasks such that the computation can proceed when the results of the fast-responding clones have returned. Recently, it has been proposed to utilize error correcting codes for straggler mitigation in distributed matrix-vector multiplication Lee *et al.*, 2018; Li *et al.*, 2016a; Dutta *et al.*, 2016; Maity *et al.*, 2018. The main idea is to partition the data matrix into K batches, and then generate N coded batches using the maximum-distance-separable (MDS) code Lin and Costello, 2004, and assign multiplication with each of the coded batches to a worker node. Benefiting from the "any K of N" property of the MDS code, the computation can be accomplished as long as *any* K fastest nodes have finished their computations, providing the system the robustness to up to $N - K$ arbitrary stragglers. This coded approach was shown to significantly outperform the state-of-the-art cloning approaches in straggler mitigation capability, and minimize the the overall computation latency.

Our first contribution on this topic is the development of optimal codes, named *polynomial codes*, to deal with stragglers in distributed high-dimensional matrix-matrix multiplication. More specifically, we consider a distributed matrix multiplication problem where we aim to compute $C = A^\top B$ from input matrices A and B. The computation is carried out using a distributed system with a master node and N worker nodes that can each stores a fixed fraction of A and B respectively (possibly in a coded manner). For this problem, we aim to design computation strategies that achieve the minimum possible *recovery threshold*, which is defined as the minimum number of workers that the master needs to wait for in order to compute C. While the prior works, i.e., the *one dimensional MDS code* (*1D MDS code*) in Lee *et al.*, 2018, and the *product code* in Lee *et al.*, 2017 apply MDS codes on the data matrices, they are sub-optimal in minimizing the recovery threshold. The main novelty and advantage of the proposed polynomial code is that, by carefully designing the algebraic structure of the coded storage at each worker, we create an MDS structure on the *intermediate computations*, instead of only the coded data matrices. This allows polynomial code to achieve order-wise improvement over state of the arts (see Table 1.1). We also prove the optimality of polynomial code by showing that it achieves the information-theoretic lower bound on the recovery threshold. As a by-product, we also prove the optimality of polynomial code under several other performance metrics considered in previous literature.

	1D MDS code	Product code	Polynomial code
Recovery threshold	$\Theta(N)$	$\Theta(\sqrt{N})$	$\Theta(1)$

Table 1.1: Comparison of recovery threshold for distributed high-dimensional matrix multiplication, over a system consisting of a master node, and N worker nodes.

Going beyond matrix algebra, we also study the straggler mitigation strategies for scenarios where the function of interest is an *arbitrary multivariate polynomial* of the input dataset. This significantly broadens the scope of the problem to cover many computations of interest in machine learning, such as various gradient and loss-function computations in learning algorithms and tensor algebraic operations (e.g., low-rank tensor approximation). In particular, we consider a computation task for which the goal is to compute a function f over a large dataset $X = (X_1, \ldots, X_K)$ to obtain K outputs $Y_1 = f(X_1), \ldots, Y_K = f(X_K)$.

The computation is carried over a system consisting of a master node and N worker nodes. Each worker i stores a coded dataset $\tilde{X}_i$ generated from X, computes $f(\tilde{X}_i)$, and sends the obtained result to the master. The master decodes the output $Y_1, \ldots, Y_K$ from the computation results of the group of the fastest workers.

For this setting, a naive repetition scheme would repeat the computation for each data block X_k onto N/K workers, yielding a recovery threshold of $N - N/K + 1 = \Theta(N)$. We propose the "Lagrange Coded Computing" (LCC) framework to minimize the recovery threshold. In particular, denoting the degree of the function f as $\deg f$, LCC promises the recovery of all output results at the master as soon as it receives computation results from $(K-1)\deg f+1$ workers. That is, LCC achieves a recovery threshold of $(K-1)\deg f+1$. Note that the recovery threshold of LCC is $\Theta(K)$, which is independent of the total number of workers N. Hence, as the network expands (i.e., N grows), compared with the naive repetition scheme, LCC benefits much more from the abundant computation resources in alleviating the negative effects caused by slow or failed nodes, which leads to a much lower computation latency. In fact, we demonstrate through proving a matching information-theoretic converse that LCC achieves the minimum possible recovery threshold among all distributed computing schemes.

The key idea of LCC is to encode the input dataset using the well-known Lagrange interpolation polynomial, in order to create computation redundancy in a novel coded form across the workers. This redundancy can then be exploited to provide resiliency to stragglers. Additionally, we emphasize on the following two salient features of the data encoding of LCC:

- *Universal*: The data encoding is oblivious of the output function f. Therefore, the coded data placement can be performed offline without knowing which operations will be applied on the data.
- *Incremental*: When new data become available and coded data batches need to be updated, we only need to encode the new data and append them to the previously coded batches, instead of accessing the entire uncoded data and re-encoding them to update the coded data.

Finally, we specialize our general theoretical guarantees for LCC in the context of least-squares linear regression, which is one of the elemental learning tasks, and demonstrate its performance gain by optimally suppressing stragglers. Leveraging the algebraic structure of gradient computations, several strategies have been developed recently to exploit data and gradient coding for straggler mitigation in the training process (see, e.g., Lee *et al.*, 2018; Tandon *et al.*, 2017; Maity *et al.*, 2018; Karakus *et al.*, 2017; Li *et al.*, 2018c). We implement LCC for regression on Amazon EC2 clusters, and empirically compare its performance with the conventional uncoded approaches, and two state-of-the-art straggler mitigation schemes: gradient coding (GC) Tandon *et al.*, 2017; Halbawi *et al.*, 2017; Raviv *et al.*, 2017; Ye and Abbe, 2018 and matrix-vector multiplication (MVM) based approaches Lee *et al.*, 2018; Maity *et al.*, 2018. Our experiment results demonstrate that compared with the uncoded scheme, LCC improves the run-time by $6.79\times \sim 13.43\times$. Compared with the GC scheme, LCC improves the run-time by $2.36\times \sim 4.29\times$. Compared with the MVM scheme, LCC improves the run-time by $1.01\times \sim 12.65\times$.

Coding for secure and private computing

Data privacy has become a major concern in the information age. The immensity of modern datasets has popularized the use of third-party cloud services, and as a result, the threat of privacy infringement has increased dramatically. In order to alleviate this concern, techniques for private computation are essential Cramer *et al.*, 2015; Bogdanov *et al.*, 2008; Lindell, 2005; Mohassel and Zhang, 2017. Additionally, third-party service providers often have an interest in the result of the computation, and might attempt to alter it for their benefit Blanchard *et al.*, 2017b; Blanchard *et al.*, 2017a. In particular, we consider a common and important scenario where a user wishes to disperse computations over a large network of workers, subject to the following privacy and security constraints.

- *Privacy constraint*: Sets of colluding workers cannot infer anything about the input dataset in the information-theoretic sense.

- *Security constraint*: The computation must be accomplished successfully even if some workers return purposefully erroneous results.

The problem of secure and private distributed computing has been studied extensively from various perspectives in the past, mainly within the scope of secure *multiparty computation* (MPC) Cramer *et al.*, 2001; Halpern and Teague, 2004; Cramer *et al.*, 2015; Ben-Or *et al.*, 1988. Most notably, the celebrated BGW scheme Ben-Or *et al.*, 1988, which adapts the Shamir secret sharing scheme Shamir, 1979a to the realm of computation, has been a reference point for several decades. The key idea of BGW scheme is to view any computation task as composed by linear and bilinear functions to be handled in multiple rounds. It applies the Shamir secret sharing scheme to generate coded data shares with security guarantees, and computes the function on the coded shares. We generalize the proposed Lagrange Coded Computing (LCC) scheme designed for straggler mitigation purposes to also provide security and privacy guarantees to MPC systems. Specifically, similarly as before, we consider the problem of evaluating a multivariate polynomial f over dataset $X = (X_1, \ldots, X_K)$. We employ a distributed computing network with a master and N workers, and aim to compute $Y_1 = f(X_1), \ldots, Y_K = f(X_K)$. For this computing system, we propose modifications to the data encoding and computation decoding processes of LCC, and demonstrate that LCC provides a T-private and A-secure computation of f (i.e., keeping the dataset private amidst collusion of any T workers, and the computation secure amidst the presence of A Byzantine adversarial workers), for any pair (T, A) satisfying

$$N \geq (K + T - 1) \deg f + 2A + 1. \tag{1.1}$$

Furthermore, we also demonstrate that LCC achieves an optimal tradeoff between privacy and security, and requires a minimal amount of added randomness to preserve privacy.

In the presence of Byzantine workers, a subset of computation results received at the master can be arbitrarily erroneous. In order to correctly recover the computation results, during the decoding process, instead of mere polynomial interpolation, the master applies an error correcting decoding algorithm for a Reed-Solomon code of dimension $(K-1)\deg(f)+1$ and length N. This allows LCC to tolerate A malicious workers as long as $2A \le N-(K-1)\deg f - 1$. Obtaining information-theoretic privacy against colluding workers, i.e., keeping small sets of workers oblivious to the dataset does not require altering the encoding nor decoding algorithm. However, prior to encoding, the dataset X is padded by T random elements $R_1,\dots,R_T$, where T is the maximum size of sets of workers that cannot infer anything about X.

Table 1.2: Comparison between BGW based designs and LCC. The computational complexity is normalized by that of evaluating f; randomness, which refers to the number of random entries used in encoding functions, is normalized by the length of X_i.

	BGW	LCC
Complexity/worker	K	1
Frac. data/worker	1	$1/K$
Randomness	KT	T
Min. num. of workers	$2T+1$	$\deg f \cdot (K+T-1)+1$

We note from (1.1) that when $N \ge (K+T-1)\deg f + 2A + 1$, the LCC scheme *simultaneously* achieves

1. *Resiliency* against $N-((K+T-1)\deg f+2A+1)$ straggler workers that prolong computations;
2. *Security* against A malicious workers, with no computational restriction, that deliberately send erroneous data in order to affect the computation for their benefit; and
3. *(Information-theoretic) Privacy* of the dataset amidst possible collusion of up to T workers.

We also note that the number of workers the master needs to wait for does *not* scale with the total number of workers N, hence the key property of LCC is that adding one additional worker can increase its resiliency to stragglers by 1, or increase its robustness to malicious worker by 1/2, while maintaining the privacy constraint. Hence, this result essentially extends the well-known optimal scaling of error-correcting codes (i.e., adding one parity can provide robustness against 1 erasure or 1/2 error in optimal maximum distance separable codes) to the distributed computing paradigm.

Finally, compared with the state-of-the-art BGW-based designs, we show that LCC significantly improves the storage, communication, and secret-sharing overhead needed for secure and private multiparty computing (see Table 1.2).

Related works

The problem of characterizing the minimum communication for distributed computing has been previously considered in several settings in both computer science and information theory literature. In Yao, 1979, a basic computing model is proposed, where two parities have x and y and aim to compute a Boolean function $f(x, y)$ by exchanging the minimum number of bits between them. Also, the problem of minimizing the required communication for computing the modulo-two sum of distributed binary sources with symmetric joint distribution was introduced in Korner and Marton, 1979. Following these two seminal works, a wide range of communication problems in the scope of distributed computing have been studied (cf. Orlitsky and El Gamal, 1990; Becker and Wille, 1998; Kushilevitz and Nisan, 2006; Orlitsky and Roche, 2001; Nazer and Gastpar, 2007; Ramamoorthy and Langberg, 2013).

The idea of efficiently creating and exploiting *coded multicasting* for bandwidth reduction was initially proposed in the context of cache networks in Maddah-Ali and Niesen, 2014b; Maddah-Ali and Niesen, 2014a, and extended in Ji *et al.*, 2016; Karamchandani *et al.*, 2014, where caches pre-fetch part of the content in a way to enable coding during the content delivery, minimizing the network traffic. Generally speaking, we can also view the data shuffling of the considered distributed computing framework as an instance of the index coding problem Birk and Kol, 2006; Bar-Yossef *et al.*, 2011, in which a central server aims to design a broadcast message (code) with minimum length to simultaneously satisfy the requests of all the clients, given the clients' side information stored in their local caches. Note that while a randomized linear network coding approach (see e.g., Ahlswede *et al.*, 2000; Koetter and Medard, 2003;

Ho *et al.*, 2003) is sufficient to implement any multicast communication where messages are intended by all receivers, it is generally sub-optimal for index coding problems where every client requests different messages. Although the index coding problem is still open in general, for the considered distributed computing scenario where we are given the flexibility of designing Map computation (thus the flexibility of designing side information), we can prove *tight* lower bounds on the minimum communication loads, demonstrating the optimality of the proposed Coded Distributed Computing scheme.

We would like to also point out that the main focus of the index coding problem/literature is to design the optimal delivery scheme for a given (often fixed) side information at the nodes. On the other hand, the key novelty of our scheme/framework is the *design of side information (or redundant computations)* at the nodes in order to maximize the index coding (or coded multicast) opportunities. So, while index coding focused on the design of best delivery strategies, we focus on the design of best side information structure. In that sense they are complementary to each other and we can leverage any of the delivery schemes developed in the index coding literature (e.g., the schemes based on local clique cover Shanmugam *et al.*, 2013, partial and fractional clique cover Birk and Kol, 2006; Agarwal and Mazumdar, 2016, interference alignment Maleki *et al.*, 2014, and many other schemes Arbabjolfaei and Kim, 2018) in the shuffling phase.

Other than designing coded computing strategies for bandwidth reduction, there has recently been a surge of interest in developing coded computing frameworks for straggler mitigation. Initiated in Lee *et al.*, 2018, many following works has focused on designing data encoding strategies, mainly inspired by the concepts of erasure/error correcting codes for communication systems, to minimize the recovery threshold, in distributed computation of matrix-vector and matrix-matrix multiplications (e.g., Dutta *et al.*, 2016; Yu *et al.*, 2017b; Fahim *et al.*, 2017; Yu *et al.*, 2018a; Wang *et al.*, 2018a). Coded computing also finds its application in distributed machine learning, specifically for running distributed stochastic gradient descent (SGD) on a master/worker architecture. For general machine learning tasks, data encoding is not applicable due to the complicated structure of gradient computation (e.g., gradients are computed numerically using back-propagation for deep neural networks). In this scenario, “gradient coding” techniques Tandon *et al.*, 2017; Halbawi *et al.*, 2017; Raviv *et al.*, 2017; Ye and Abbe, 2018; Li *et al.*, 2018c have been designed to code across partial gradients computed from uncoded data, such that the master can recover the total gradient as the sum of all partial gradients, after receiving the computation results from the minimum possible number of workers.

The proposed Lagrange Coded Computing (LCC) scheme improves and expands these prior works in a few aspects: *Generality*–LCC significantly expands the computation class for which we know how to design coded computing to go beyond linear and bilinear computations that have so far been the main research focus. In particular, it can be applied to more general multivariate polynomial computations that arise in machine learning applications. *Universality*–once the data has been coded, any polynomial up to a certain degree can be computed distributedly via LCC. In other words, data encoding of LCC can be *universally* used for any polynomial computation. This is in stark contrast to previous task-specific coding techniques in the literature. *Security and privacy*–other than straggler mitigation, LCC also extends the application of coded computing to secure and private computing for general polynomial computations.

The security and privacy issue of distributed computing has been extensively studied in the literature of secure multiparty computing (MPC) and secure machine learning/data mining, Ben-Or *et al.*, 1988; Cramer *et al.*, 2001; Lindell, 2005; Cramer *et al.*, 2015; Halpern and Teague, 2004; Huang *et al.*, 2011. As a representative example, we briefly describe the celebrated BGW MPC scheme Ben-Or *et al.*, 1988. Given data inputs $\{X_i\}_{i=1}^K$, the problem is to compute outputs $\{f(X_i)\}_{i=1}^K$ using N workers in a privacy-preserving manner (i.e., colluding workers cannot infer anything about the dataset using their local data). To do that, BGW first uses Shamir's scheme Shamir, 1979a to encode each X_i as a polynomial $P_i(z) = X_i + Z_{i,1}z + \ldots + Z_{i,T}z^T$, where $Z_{i,j}$'s are i.i.d. uniformly random variables and T is the number of colluding workers that should be tolerated. Then, each worker ℓ stores the coded data

$\{P_i(\alpha_\ell)\}_{i=1}^K$, for a distinct α_ℓ, and computes $\{f(P_i(\alpha_\ell))\}_{i=1}^K$. Hence, for each i, each worker provides the evaluation of the degree-$(\deg f \cdot T)$ polynomial $f(P_i(z))$ at a distinct point α_ℓ. The polynomial $f(P_i(z))$ can be interpolated using computation results from $\deg f \cdot T + 1$ workers, and $f(X_i)$ is obtained by taking the constant term of $f(P_i(z))$.[1] In the proposed LCC scheme, instead of hiding X_i's individually in data encoding, we code *across* X_i's together with some added random inputs. This gives rise to significant reduction on storage overhead, computational complexity, and the amount of padded randomness. However, under the same condition, LCC scheme requires $N \geq \deg f \cdot (K + T - 1) + 1$ number of workers, which is larger than that of the BGW scheme. So, in some sense LCC achieves reduction in storage overhead, computational complexity, and the amount of padded randomness, at the expense of increasing the number of needed workers (or reducing the fraction of Byzantine workers that can be tolerated). We refer to Table 1.2 for a detailed comparion between BGW and LCC.

Coding techniques have been recently developed to provide security and privacy guarantees to distributed computing. Specifically, staircase codes Bitar *et al.*, 2018 were proposed to combat stragglers in linear computations (e.g., matrix-vector multiplications) while preserving data privacy, improving the computation latency of the conventional secure computing schemes based on secret sharing Shamir, 1979a; McEliece and Sarwate, 1981. The proposed LCC scheme generalizes the staircase codes beyond linear computations. Even for the linear case, LCC guarantees data privacy against T colluding workers by introducing less randomness than Bitar *et al.*, 2018 (T rather than $TK/(K-T)$). Beyond linear computations, a recent work Nodehi and Maddah-Ali, 2018 has combined ideas from the BGW scheme and the polynomial code Yu *et al.*, 2017b to form *polynomial sharing*, a private coded computing scheme for arbitrary matrix polynomials. However, polynomial sharing inherits the undesired BGW property of performing a communication round for *every* bilinear operation in the polynomial; a feature that drastically

[1] It is also possible to use the conventional multi-round BGW, which only requires $N \geq 2T + 1$ workers to ensure T-privacy. However, multiple rounds of computation and communication ($\Omega(\log(\deg f))$ rounds) are needed, which further increases its communication overhead.

reduces communication efficiency, and is circumvented by the one-shot approach of LCC. *DRACO* Chen *et al.*, 2018 was proposed as a secure distributed training algorithm that is robust to Byzantine workers. Since DRACO is designed for general gradient computations, it employs a blackbox approach, i.e., the coding is applied on the gradients computed from uncoded data, but not on the data itself, which is similar to the gradient coding techniques Tandon *et al.*, 2017; Halbawi *et al.*, 2017; Raviv *et al.*, 2017; Ye and Abbe, 2018; Li *et al.*, 2018c designed primarily for stragglers. For this approach, Chen *et al.*, 2018 show that a $2A + 1$ *multiplicative* factor of redundant computations is needed to be robust to A Byzantine workers. For the proposed LCC however, the blackbox approach is disregarded in favor of an algebraic one, and consequently, a $2A$ *additive* factor suffices.

2

Coding for Bandwidth Reduction

In this chapter, we focus on a general distributed computing framework, motivated by prevalent structures like MapReduce Dean and Ghemawat, 2004 and Spark Zaharia *et al.*, 2010, in which the overall computation is decomposed into two stages: "Map" and "Reduce". Firstly in the Map stage, distributed computing nodes process parts of the input data locally, generating some intermediate values according to their designed Map functions. Next, they exchange the calculated intermediate values among each other (a.k.a. data shuffling), in order to calculate the final output results distributedly using their designed Reduce functions.

Within this framework, data shuffling often appears to limit the performance of distributed computing applications, including self-join Ahmad *et al.*, 2012, TeraSort Guo *et al.*, 2013, and machine learning algorithms Chowdhury *et al.*, 2011. For example, in a Facebook's Hadoop cluster, it is observed that 33% of the overall job execution time is spent on data shuffling Chowdhury *et al.*, 2011. Also as is observed in Zhang *et al.*, 2013, 70% of the overall job execution time is spent on data shuffling when running self-join on an Amazon EC2 cluster *Amazon Elastic Compute Cloud (EC2)* n.d. This bottleneck becomes even worse when training deep neural networks (e.g., ResNet-50 He *et al.*, 2016) on distributed computing systems, where partial gradients with millions of entries are shuffled across networks to update model parameters Chilimbi *et al.*, 2014.

As such motivated, we ask this fundamental question that *if coding can help distributed computing in reducing the load of communication and speeding up the overall computation?* Coding is known to be helpful in coping with the channel uncertainty in telecommunication systems and also in reducing the storage cost in distributed storage systems and cache networks. In this chapter, we extend the application of coding to *distributed computing* and propose a framework to substantially reduce the load of data shuffling via coding and some extra computing in the Map phase.

More specifically, we first formalize a MapReduce-type distributed computing framework, and define the "computation load" as the amount of local Map computations performed at distributed computing nodes, and the "communication load" as the amount of information bits shuffled between nodes. We characterize a fundamental "inversely proportional" tradeoff relationship between computation load and communication load. In particular, we propose a coded computing scheme, named "Coded Distributed Computing" (CDC), which demonstrates that increasing the computation load of the Map phase by a factor of r (i.e., evaluating each Map function at r *carefully chosen* nodes) can create novel coding opportunities in the data shuffling phase that reduce the communication load by the same factor. We also show that CDC is optimal, in the sense that it achieves the best tradeoff between computation and communication in the proposed MapReduce framework.

Having theoretically characterized the tradeoff between computation and communication, we exploit this tradeoff to improve the run-time performance of practical workloads. Particularly, we apply the principles of the CDC scheme to `TeraSort` *Hadoop TeraSort* n.d., which is a widely used benchmark in Hadoop MapReduce *Apache Hadoop* n.d. for distributedly sorting terabytes of data O'Malley, 2008, and develop a new distributed sorting algorithm, named `CodedTeraSort`, which imposes *structured* redundancy in data to enable coding opportunities for efficient data shuffling. We empirically demonstrate that `CodedTeraSort` speeds up the state-of-the-art sorting algorithms by 1.97×- 3.39× in typical settings of interest.

Having demonstrated the impact of coding on improving the performance of applications run on wired networks like datacenters, we also introduce the concept of coded computing to tackle the scenarios of mobile edge/fog computing, where the communication bottleneck is even more severe due to the low data rate and the large number of mobile users. In particular, we consider a wireless distributed computing platform, which is composed of a cluster of memory-limited mobile users and an access point at the network edge. The users collaborate with each other through the access point to satisfy their computational needs that require processing a large dataset. For this platform we propose a coded wireless distributed computing (CWDC) scheme that jointly designs the local storage and computation for each user, and the communication between users through the access point. The CWDC scheme achieves a constant bandwidth consumption that is independent of the number of users in the system, which leads to a *scalable* design of the platform that can simultaneously accommodate an arbitrary number of users.

Finally, we end this chapter with some related works and open problems along this research direction.

2.1 A fundamental tradeoff between computation and communication

In this section, we formulate a general distributed computing framework motivated by MapReduce, and characterize the optimal tradeoff between computation and communication within this framework.

2.1.1 Problem formulation: a distributed computing framework

We consider the problem of computing Q arbitrary output functions from N input files using a cluster of K distributed computing nodes (servers), for some positive integers $Q, N, K \in \mathbb{N}$, with $N \geq K$.[1] More specifically,

[1]The motivation for considering simultaneous computation of Q functions is that we consider a common scenario in which many computation requests (over the same dataset) are continuously submitted (e.g., database queries, web search, loss computation in machine learning, etc).

given N input files $w_1, \ldots, w_N \in \mathbb{F}_{2^F}$, for some $F \in \mathbb{N}$, the goal is to compute Q output functions $\phi_1, \ldots, \phi_Q$, where $\phi_q : (\mathbb{F}_{2^F})^N \to \mathbb{F}_{2^B}$, $q \in \{1, \ldots, Q\}$, maps all input files to an output $u_q = \phi_q(w_1, \ldots, w_N) \in \mathbb{F}_{2^B}$, for some $B \in \mathbb{N}$.

Motivated by MapReduce, we assume that as illustrated in Figure 2.1 the computation of the output function ϕ_q, $q \in \{1, \ldots, Q\}$ can be decomposed as follows:

$$\phi_q(w_1, \ldots, w_N) = h_q(g_{q,1}(w_1), \ldots, g_{q,N}(w_N)), \tag{2.1}$$

where

- The "Map" functions $\vec{g}_n = (g_{1,n}, \ldots, g_{Q,n}) : \mathbb{F}_{2^F} \to (\mathbb{F}_{2^T})^Q$, $n \in \{1, \ldots, N\}$, maps the input file w_n into Q length-T *intermediate values* $v_{q,n} = g_{q,n}(w_n) \in \mathbb{F}_{2^T}$, $q \in \{1, \ldots, Q\}$, for some $T \in \mathbb{N}$.[2]
- The "Reduce" functions $h_q : (\mathbb{F}_{2^T})^N \to \mathbb{F}_{2^B}$, $q \in \{1, \ldots, Q\}$, maps the intermediate values of the output function ϕ_q in all input files into the output value $u_q = h_q(v_{q,1}, \ldots, v_{q,N})$.

Remark 2.1. Note that for every set of output functions $\phi_1, \ldots, \phi_Q$ such a Map-Reduce decomposition exists (e.g., setting $g_{q,n}'s$ to identity functions such that $g_{q,n}(w_n) = w_n$ for all $n = 1, \ldots, N$, and h_q to ϕ_q in (2.1)). However, such a decomposition is not unique, and in the distributed computing literature, there has been quite some work on developing appropriate decompositions of computations like join, sorting and matrix multiplication (see, e.g., Dean and Ghemawat, 2004; Rajaraman and Ullman, 2011), for them to be performed efficiently in a distributed manner. Here we do not impose any constraint on how the Map and Reduce functions are chosen (for example, they can be arbitrary linear or non-linear functions). □

[2]When mapping a file, we compute Q intermediate values in parallel, one for each of the Q output functions. The main reason to do this is that parallel processing can be efficiently performed for applications that fit into the MapReduce framework. In other words, mapping a file according to one function is only marginally more expensive than mapping according to all functions. For example, for the canonical Word Count task, while we are scanning a document to count the number of appearances of one word, we can simultaneously count the numbers of appearances of other words with marginally increased computation cost.

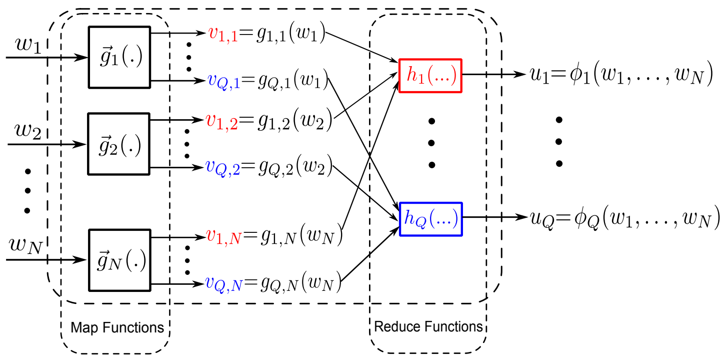


Figure 2.1: Illustration of a two-stage distributed computing framework. The overall computation is decomposed into computing a set of Map and Reduce functions.

The above computation is carried out by K distributed computing nodes, labelled as Node $1,\dots,$Node K. They are interconnected through a multicast network. Following the above decomposition, the computation proceeds in three phases: *Map*, *Shuffle* and *Reduce*.

Map Phase: Node k, $k \in \{1,\dots,K\}$, computes the Map functions of a set of files $\mathcal{M}_k$, which are stored on Node k, for some design parameter $\mathcal{M}_k \subseteq \{w_1,\dots,w_N\}$. For each file w_n in $\mathcal{M}_k$, Node k computes $\vec{g}_n(w_n) = (v_{1,n},\dots,v_{Q,n})$. We assume that each file is mapped by at least one node, i.e., $\underset{k=1,\dots,K}{\cup} \mathcal{M}_k = \{w_1,\dots,w_N\}$.

Definition 2.1 (Computation Load). We define the *computation load*, denoted by r, $1 \le r \le K$, as the total number of Map functions computed across the K nodes, normalized by the number of files N, i.e., $r \triangleq \frac{\sum_{k=1}^{K} |\mathcal{M}_k|}{N}$. The computation load r can be interpreted as the average number of nodes that map each file. $\Diamond$

Shuffle Phase: Node k, $k \in \{1,\dots,K\}$, is responsible for computing a subset of output functions, whose indices are denoted by a set $\mathcal{W}_k \subseteq \{1,\dots,Q\}$. We focus on the case $\frac{Q}{K} \in \mathbb{N}$, and utilize a *symmetric* task assignment across the K nodes to maintain load balance. More precisely, we require 1) $|\mathcal{W}_1| = \cdots = |\mathcal{W}_K| = \frac{Q}{K}$, 2) $\mathcal{W}_j \cap \mathcal{W}_k = \varnothing$ for all $j \neq k$.

Remark 2.2. Beyond the symmetric task assignment considered in this chapter, characterizing the optimal computation-communication tradeoff allowing general asymmetric task assignments is a challenging open problem. As the first step to study this problem, the follow-up work Yu *et al.*, 2017a considers the scenario in which the number of output functions Q is fixed and the computing resources are abundant (e.g., number of computing nodes $K \gg Q$), it is shown in Yu *et al.*, 2017a that asymmetric task assignments can do better than the symmetric ones in optimizing the overall run-time performance. $\square$

To compute the output u_q for some $q \in \mathcal{W}_k$, Node k needs the intermediate values that are *not* computed *locally* in the Map phase, i.e., $\{v_{q,n} : q \in \mathcal{W}_k, w_n \notin \mathcal{M}_k\}$. After Node k, $k \in \{1, \dots, K\}$, has finished mapping all the files in $\mathcal{M}_k$, the K nodes proceed to exchange the needed intermediate values. In particular, each node k creates an input symbol $X_k \in \mathbb{F}_{2^{\ell_k}}$, for some $\ell_k \in \mathbb{N}$, as a function of the intermediate values computed locally during the Map phase, i.e., for some encoding function $\psi_k : (\mathbb{F}_{2^T})^{Q|\mathcal{M}_k|} \to \mathbb{F}_{2^{\ell_k}}$ at Node k, we have

$$X_k = \psi_k\left(\{\vec{g}_n : w_n \in \mathcal{M}_k\}\right). \tag{2.2}$$

Having generated the message X_k, Node k multicasts it to all other nodes.

By the end of the Shuffle phase, each of the K nodes receives $X_1, \dots, X_K$ free of error.

Definition 2.2 (Communication Load). We define the *communication load*, denoted by L, $0 \le L \le 1$, as $L \triangleq \frac{\ell_1 + \cdots + \ell_K}{QNT}$. That is, L represents the (normalized) total number of bits communicated by the K nodes during the Shuffle phase.[3] $\Diamond$

Reduce Phase: Node k, $k \in \{1, \dots, K\}$, uses the messages $X_1, \dots, X_K$ communicated in the Shuffle phase, and the local results from the Map phase $\{\vec{g}_n : w_n \in \mathcal{M}_k\}$ to construct inputs to the corresponding Reduce

[3]For notational convenience, we define all variables in binary extension fields. However, one can consider arbitrary field sizes. For example, we can consider all intermediate values $v_{q,n}$, $q = 1, \dots, Q$, $n = 1, \dots, N$, to be in the field $\mathbb{F}_{p^T}$, for some prime number p and positive integer T, and the symbol communicated by Node k (i.e., X_k), to be in the field $\mathbb{F}_{s^{\ell_k}}$ for some prime number s and positive integer ℓ_k, for all $k = 1, \dots, K$. In this case, the communication load can be defined as $L \triangleq \frac{(\ell_1 + \cdots + \ell_K)\log s}{QNT \log p}$.

functions of $\mathcal{W}_k$, i.e., for each $q \in \mathcal{W}_k$ and some decoding function $\chi_k^q : \mathbb{F}_{2^{\ell_1}} \times \cdots \times \mathbb{F}_{2^{\ell_K}} \times (\mathbb{F}_{2^T})^{Q|\mathcal{M}_k|} \to (\mathbb{F}_{2^T})^N$, Node k computes

$$(v_{q,1}, \ldots, v_{q,N}) = \chi_k^q \left(X_1, \ldots, X_K, \{\vec{g}_n : w_n \in \mathcal{M}_k\}\right). \tag{2.3}$$

Finally, Node k, $k \in \{1, \ldots, K\}$, computes the Reduce function $u_q = h_q(v_{q,1} \ldots v_{q,N})$ for all $q \in \mathcal{W}_k$.

We say that a computation-communication pair $(r, L) \in \mathbb{R}^2$ is *feasible* if for any $\delta > 0$ and sufficiently large N, there exist $\mathcal{M}_1, \ldots, \mathcal{M}_K$, $\mathcal{W}_1, \ldots, \mathcal{W}_K$, a set of encoding functions $\{\psi_k\}_{k=1}^K$, and a set of decoding functions $\{\chi_k^q : q \in \mathcal{W}_k\}_{k=1}^K$ that achieve a computation-communication pair $(\tilde{r}, \tilde{L}) \in \mathbb{Q}^2$ such that $|r - \tilde{r}| \leq \delta$, $|L - \tilde{L}| \leq \delta$, and Node k can successfully compute all the output functions whose indices are in $\mathcal{W}_k$, for all $k \in \{1, \ldots, K\}$.

Definition 2.3. We define the *computation-communication function* of the distributed computing framework

$$L^*(r) \triangleq \inf\{L : (r, L) \text{ is feasible}\}. \tag{2.4}$$

$L^*(r)$ characterizes the optimal tradeoff between computation and communication in this framework. $\diamondsuit$

Example (uncoded scheme). In the Shuffle phase of a simple "uncoded" scheme, each node receives the needed intermediate values sent uncodedly by some other nodes. Since a total of QN intermediate values are needed across the K nodes and $rN \cdot \frac{Q}{K} = \frac{rQN}{K}$ of them are already available after the Map phase, the communication load achieved by the uncoded scheme

$$L_{\text{uncoded}}(r) = 1 - r/K. \tag{2.5}$$

2.1.2 Main Results

Theorem 2.1. The computation-communication function of the distributed computing framework, $L^*(r)$ is given by

$$L^*(r) = L_{\text{coded}}(r) \triangleq \tfrac{1}{r} \cdot (1 - \tfrac{r}{K}), \quad r \in \{1, \ldots, K\}, \tag{2.6}$$

for sufficiently large T. For general $1 \leq r \leq K$, $L^*(r)$ is the lower convex envelop of the above points $\{(r, \frac{1}{r} \cdot (1 - \frac{r}{K})) : r \in \{1, \ldots, K\}\}$.

We prove the achievability of Theorem 2.1 by proposing a *coded* scheme, named Coded Distributed Computing, in Section 2.1.3. We demonstrate that no other scheme can achieve a communication load smaller than the lower convex envelop of the points $\{(r, \frac{1}{r} \cdot (1 - \frac{r}{K})) : r \in \{1, \dots, K\}\}$ by proving the converse in Section 2.1.4.

Remark 2.3. Theorem 2.1 exactly characterizes the optimal tradeoff between the computation load and the communication load in the considered distributed computing framework. □

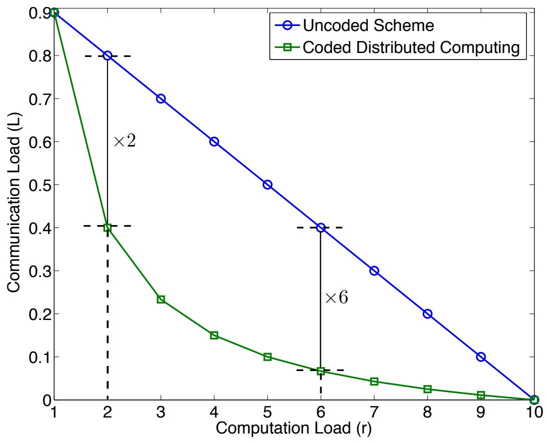


Figure 2.2: Comparison of the communication load achieved by the proposed coded scheme in Theorem 2.1 with that of the uncoded scheme in (2.5), for $Q = 10$ output functions, $N = 2520$ input files and $K = 10$ computing nodes.

Remark 2.4. For $r \in \{1, \dots, K\}$, the communication load achieved in Theorem 2.1 is less than that of the uncoded scheme in (2.5) by a multiplicative factor of r, which equals the computation load and can grow unboundedly as the number of nodes K increases if e.g., $r = \Theta(K)$. As illustrated in Figure 2.2, while the communication load of the uncoded scheme decreases linearly as the computation load increases, $L_{\text{coded}}(r)$ achieved in Theorem 2.1 is inversely proportional to the computation load. □

Remark 2.5. While increasing the computation load r causes a longer Map phase, the coded achievable scheme of Theorem 2.1 maximizes the reduction of the communication load using the extra computations. Therefore, Theorem 2.1 provides an analytical framework to optimally trading the computation power in the Map phase for more bandwidth in the Shuffle phase, which helps to minimize the overall execution time of applications whose performances are limited by data shuffling. In the next section, we will empirically demonstrate this idea through experiments on a widely-used practical workload. □

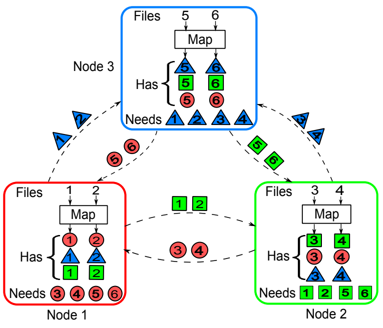
Remark 2.6. In Li *et al.*, 2018b, we also consider a generalization of the above distributed computing framework, which we call “cascaded distributed computing framework”, where after the Map phase, each Reduce function is computed by $s > 1$ nodes. This generalized model is motivated by the fact that many distributed computing jobs require multiple rounds of Map and Reduce executions, where the Reduce results of the previous round serve as the inputs to the Map functions of the next round. For the cascaded distributed computing framework, we generalize our coded computing scheme to achieve the optimal tradeoff between computation and communication loads. □

2.1.3 Coded Distributed Computing

In this subsection, we formally prove the upper bound in Theorem 2.1 by describing and analyzing the Coded Distributed Computing (CDC) scheme. Before we present the general CDC scheme, we first illustrate its key coding ideas via an example.

Illustrative example

We consider a MapReduce-type problem in Figure 2.3 for distributed computing of $Q = 3$ output functions, represented by red/circle, green/square, and blue/triangle respectively, from $N = 6$ input files, using $K = 3$ computing nodes. Nodes 1, 2, and 3 are respectively responsible for final reduction of red/circle, green/square, and blue/triangle output functions. We first consider the case where no redundancy is imposed on the computations, i.e., each file is mapped once and computation load $r = 1$. As shown in Figure 2.3(a), Node k maps File $2k - 1$ and File $2k$ for $k = 1, 2, 3$. In this case, each node maps 2 input files


(a) Uncoded Distributed Computing Scheme.

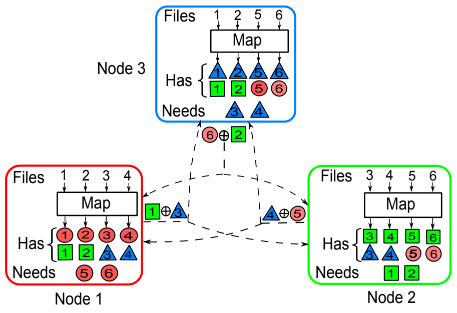

(b) Coded Distributed Computing Scheme.

Figure 2.3: Illustrations of the conventional uncoded distributed computing scheme with computation load $r = 1$, and the proposed Coded Distributed Computing scheme with computation load $r = 2$, for computing $Q = 3$ functions from $N = 6$ inputs on $K = 3$ nodes.

locally. In Figure 2.3, we represent, for example, the intermediate value of the red/circle function in File n using a red circle labelled by n, for all $n = 1, \ldots, 6$. Similar representations follow for the green/square and the blue/triangle functions. After the Map phase, each node obtains 2 out of 6 required intermediate values to reduce the output function it is responsible for (e.g., Node 1 knows the red circles in File 1 and File 2). Hence, each node needs 4 intermediate values from the other nodes, yielding a communication load of $\frac{4\times3}{3\times6} = \frac{2}{3}$.

Now, we demonstrate how the proposed CDC scheme trades the computation load to slash the communication load via in-network coding. As shown in Figure 2.3(b), we double the computation load such that each file is now mapped on two nodes ($r = 2$). It is apparent that since more local computations are performed, each node now only requires 2 other intermediate values, and an uncoded shuffling scheme would achieve a communication load of $\frac{2\times3}{3\times6} = \frac{1}{3}$. However, we can do much better with coding. As shown in Figure 2.3(b), instead of unicasting individual intermediate values, every node multicasts a bit-wise XOR, denoted by $\oplus$, of 2 locally computed intermediate values to the other two nodes, simultaneously satisfying their data demands. For example, knowing the blue/triangle in File 3, Node 2 can cancel it from the coded packet sent by Node 1, recovering the needed green/square in File 1. Therefore, this coding incurs a communication load of $\frac{3}{3\times 6} = \frac{1}{6}$, achieving a $2\times$ gain over the uncoded shuffling.

General CDC scheme

We first consider the integer-valued computation load $r \in \{1, \dots, K\}$, and then generalize the CDC scheme for any $1 \le r \le K$. When $r = K$, every node can map all the input files and compute all the output functions locally, thus no communication is needed and $L^*(K) = 0$. In what follows, we focus on the case where $r < K$.

We consider sufficiently large number of input files N, and $\binom{K}{r}(\eta - 1) < N \le \binom{K}{r}\eta$, for some $\eta \in \mathbb{N}$. We first inject $\binom{K}{r}\eta - N$ empty files into the system to obtain a total of $\bar{N} = \binom{K}{r}\eta$ files, which is now a multiple of of $\binom{K}{r}$. We note that $\lim\limits_{N\to\infty} \frac{\bar{N}}{N} = 1$. Next, we proceed to present the CDC scheme for a system with $\bar{N}$ input files $w_1, \dots, w_{\bar{N}}$.

Map phase design. The $\bar{N}$ input files are evenly partitioned into $\binom{K}{r}$ disjoint batches of size η, each corresponding to a subset $\mathcal{T} \subset \{1, \dots, K\}$ of size r, i.e.,

$$\{w_1, \dots, w_{\bar{N}}\} = \bigcup_{\mathcal{T}\subset\{1,\dots,K\}, |\mathcal{T}|=r} \mathcal{B}_{\mathcal{T}}, \tag{2.7}$$

where $\mathcal{B}_{\mathcal{T}}$ denotes the batch of η files corresponding to the subset $\mathcal{T}$.

Given this partition, Node k, $k \in \{1, \dots, K\}$, computes the Map functions of the files in $\mathcal{B}_{\mathcal{T}}$ if $k \in \mathcal{T}$. Or equivalently, $\mathcal{B}_{\mathcal{T}} \subseteq \mathcal{M}_k$ if $k \in \mathcal{T}$. Since each node is in $\binom{K-1}{r-1}$ subsets of size r, each node computes $\binom{K-1}{r-1}\eta = \frac{r\bar{N}}{K}$ Map functions, i.e., $|\mathcal{M}_k| = \frac{r\bar{N}}{K}$ for all $k \in \{1, \dots, K\}$. After the Map phase, Node k, $k \in \{1, \dots, K\}$, knows the intermediate values of all Q output functions in the files in $\mathcal{M}_k$, i.e., $\{v_{q,n} : q \in \{1, \dots, Q\}, w_n \in \mathcal{M}_k\}$.

Coded data shuffling. We focus on the case where the number of the output functions Q satisfies $\frac{Q}{K} \in \mathbb{N}$, and enforce a symmetric assignment of the Reduce functions such that every node reduces $\frac{Q}{K}$ functions. That is, $|\mathcal{W}_1| = \cdots = |\mathcal{W}_K| = \frac{Q}{K}$, and $\mathcal{W}_j \cap \mathcal{W}_k = \varnothing$ for all $j \ne k$.

For any subset $\mathcal{P} \subset \{1, \dots, K\}$, and $k \notin \mathcal{P}$, we denote the set of intermediate values needed by Node k and known *exclusively* by nodes whose indices are in $\mathcal{P}$ as $\mathcal{V}^k_{\mathcal{P}}$. More formally:

$$\mathcal{V}^k_{\mathcal{P}} \triangleq \{v_{q,n} : q \in \mathcal{W}_k, w_n \in \bigcap_{i\in\mathcal{P}} \mathcal{M}_i, w_n \notin \bigcup_{i\notin\mathcal{P}} \mathcal{M}_i\}. \tag{2.8}$$

For each subset $\mathcal{S} \subseteq \{1,\ldots,K\}$ of size $|\mathcal{S}| = r+1$, we perform the following three steps to shuffle the intermediate results.

- **Step 1: data association.** For each $k \in \mathcal{S}$, $\mathcal{V}^k_{\mathcal{S}\backslash\{k\}}$ is the set of intermediate values that are requested by Node k and are computed from the files in the batch $\mathcal{B}_{\mathcal{S}\backslash\{k\}}$, and they are exclusively known at all nodes whose indices are in $\mathcal{S}\backslash\{k\}$. We evenly and arbitrarily split $\mathcal{V}^k_{\mathcal{S}\backslash\{k\}}$, into r disjoint segments $\{\mathcal{V}^k_{\mathcal{S}\backslash\{k\},i} : i \in \mathcal{S}\backslash\{k\}\}$, where $\mathcal{V}^k_{\mathcal{S}\backslash\{k\},i}$ denotes the segment associated with Node i in $\mathcal{S}\backslash\{k\}$ for Node k. That is, $\mathcal{V}^k_{\mathcal{S}\backslash\{k\}} = \bigcup_{i\in\mathcal{S}\backslash\{k\}} \mathcal{V}^k_{\mathcal{S}\backslash\{k\},i}$.

- **Step 2: coded multicast.** Each node i, $i \in \mathcal{S}$, computes the bit-wise XOR, denoted by $\oplus$, of all the segments associated with it in $\mathcal{S}$, generating a coded segment $X^{\mathcal{S}}_i = \bigoplus_{k\in\mathcal{S}\backslash\{i\}} \mathcal{V}^k_{\mathcal{S}\backslash\{k\},i}$. Then, Node i multicasts $X^{\mathcal{S}}_i$ to all other nodes in $\mathcal{S} \setminus \{i\}$.

- **Step 3: decoding.** Having received $X^{\mathcal{S}}_i$ from Node i, Node k computes the bit-wise XOR of $X^{\mathcal{S}}_i$ with its local data segments $\{\mathcal{V}^j_{\mathcal{S}\backslash\{j\},i} : j \in \mathcal{S} \setminus \{i,k\}\}$ to recover $\mathcal{V}^k_{\mathcal{S}\backslash\{k\},i} = \bigoplus_{j\in\mathcal{S}\backslash\{i,k\}} \mathcal{V}^j_{\mathcal{S}\backslash\{j\},i} \oplus X^{\mathcal{S}}_i$. Having decoded the data segments $\mathcal{V}^k_{\mathcal{S}\backslash\{k\},i}$ for all $i \in \mathcal{S} \setminus \{k\}$, Node k concatenates them to recover $\mathcal{V}^k_{\mathcal{S}\backslash\{k\}}$.

After we iterate the above data shuffling process over all subsets of $r+1$ nodes, it is easy to see that for each node k, other than its locally computed intermediate values, it has recovered all the required intermediate values, i.e., $\{\mathcal{V}^k_{\mathcal{S}\backslash\{k\}} : \mathcal{S} \subseteq \{1,\ldots,K\}, |\mathcal{S}| = r+1, k \in \mathcal{S}\}$, to compute the Reduce functions locally.

Communication load. Since the coded segment $X^{\mathcal{S}}_i$ has a size of $\frac{Q}{K} \cdot \frac{\eta T}{r}$ bits for each $i \in \mathcal{S}$, there are a total of $\frac{Q}{K} \cdot \frac{\eta T}{r}(r+1)$ bits shuffled across the network in each subset $\mathcal{S}$ of size $r+1$. Therefore, the communication load achieved by this coded data shuffling scheme, for $r \in \{1, \ldots, K-1\}$, is

$$L_{\text{coded}}(r) = \lim_{N \to \infty} \frac{\binom{K}{r+1} \frac{Q}{K} \cdot \frac{\eta T}{r} (r+1)}{QNT} = \lim_{N \to \infty} \frac{\bar{N}(K-r)}{NKr}$$
$$= \tfrac{1}{r} \cdot (1 - \tfrac{r}{K}) \tag{2.9}$$

Non-integer valued computation load. For non-integer valued computation load $r \geq 1$, we generalize the CDC scheme as follows. We first expand the computation load $r = \alpha r_1 + (1-\alpha) r_2$ as a convex combination of $r_1 \triangleq \lfloor r \rfloor$ and $r_2 \triangleq \lceil r \rceil$, for some $0 \leq \alpha \leq 1$. Then we partition the set of $\bar{N}$ input files $\{w_1, \ldots, w_{\bar{N}}\}$ into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$ of sizes $|\mathcal{I}_1| = \alpha \bar{N}$ and $|\mathcal{I}_2| = (1-\alpha)\bar{N}$. We next apply the CDC scheme described above respectively to the files in $\mathcal{I}_1$ with a computation load r_1 and the files in $\mathcal{I}_2$ with a computation load r_2, to compute each of the Q output functions at the same node. This results in a communication load of

$$\lim_{N \to \infty} \frac{Q\alpha \bar{N} L_{\text{coded}}(r_1) T + Q(1-\alpha)\bar{N} L_{\text{coded}}(r_2) T}{QNT}$$
$$= \alpha L_{\text{coded}}(r_1) + (1-\alpha) L_{\text{coded}}(r_2), \tag{2.10}$$

where $L_{\text{coded}}(r)$ is the communication load achieved by CDC in (2.9) for integer-valued r.

Using this generalized CDC scheme, for any two integer-valued computation loads r_1 and r_2, the points on the line segment connecting $(r_1, L_{\text{coded}}(r_1))$ and $(r_2, L_{\text{coded}}(r_2))$ are achievable. Therefore, for general $1 \leq r \leq K$, the *lower convex envelop* of the achievable points $\{(r, L_{\text{coded}}(r)) : r \in \{1, \ldots, K\}\}$ is achievable. This proves the upper bound on the computation-communication function in Theorem 2.1.

Remark 2.7. The ideas of efficiently creating and exploiting coded multicasting opportunities have been introduced in caching problems Maddah-Ali and Niesen, 2014b; Maddah-Ali and Niesen, 2014a; Ji *et al.*, 2016. Through the above description of the CDC scheme, we illustrated how coding opportunities can be utilized in distributed computing to slash the load of communicating intermediate values, by designing a particular assignment of extra computations across distributed computing nodes. We note that the calculated intermediate values in the Map phase mimics the locally stored cache contents in caching problems, providing the “side information” to enable coding in the following Shuffle phase (or content delivery). □

Remark 2.8. Generally speaking, we can view the Shuffle phase of the considered distributed computing framework as an instance of the index coding problem Birk and Kol, 2006; Bar-Yossef *et al.*, 2011, in which a central server aims to design a broadcast message (code) with minimum length to simultaneously satisfy the requests of all the clients, given the clients’ side information stored in their local caches. Note that while a randomized linear network coding approach (see, e.g., Ahlswede *et al.*, 2000; Koetter and Medard, 2003; Ho *et al.*, 2003) is sufficient to implement any multicast communication where messages are intended by all receivers, it is generally sub-optimal for index coding problems where every client requests different messages. Although the index coding problem is still open in general, for the considered distributed computing scenario where we are given the flexibility of designing Map computation (thus the flexibility of designing side information), we next prove *tight* lower bounds on the minimum communication load, demonstrating the optimality of the proposed CDC scheme. □

2.1.4 Optimality of CDC

In this subsection, we prove the lower bound on $L^*(r)$ in Theorem 2.1, and demonstrate the optimality of CDC in minimizing the communication load.

For $k \in \{1, \ldots, K\}$, we denote the set of indices of the files mapped by Node k as $\mathcal{M}_k$, and the set of indices of the Reduce functions computed by Node k as $\mathcal{W}_k$. As the first step, we consider the communication load for a given file assignment $\mathcal{M} \triangleq (\mathcal{M}_1, \mathcal{M}_2 \ldots, \mathcal{M}_K)$ in the Map phase. We denote the minimum communication load under the file assignment $\mathcal{M}$ by $L^*_{\mathcal{M}}$.

We denote the number of files that are mapped at j nodes under a file assignment $\mathcal{M}$, as $a^j_{\mathcal{M}}$, for all $j \in \{1, \ldots, K\}$:

$$a^j_{\mathcal{M}} = \sum_{\mathcal{J} \subseteq \{1,\ldots,K\}: |\mathcal{J}|=j} |(\bigcap_{k\in\mathcal{J}} \mathcal{M}_k) \backslash (\bigcup_{i\notin\mathcal{J}} \mathcal{M}_i)|. \tag{2.11}$$

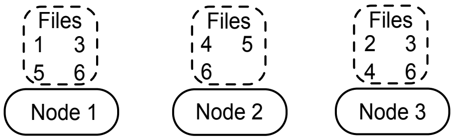


Figure 2.4: A file assignment for $N = 6$ files and $K = 3$ nodes.

For example, for the particular file assignment in Figure 2.4, i.e., $\mathcal{M} = (\{1,3,5,6\},\{4,5,6\},\{2,3,4,6\})$, $a^1_{\mathcal{M}} = 2$ since File 1 and File 2 are mapped on a single node (i.e., Node 1 and Node 3 respectively). Similarly, we have $a^2_{\mathcal{M}} = 3$ (Files 3, 4, and 5), and $a^3_{\mathcal{M}} = 1$ (File 6).

For a particular file assignment $\mathcal{M}$, we present a lower bound on $L^*_{\mathcal{M}}$ in the following lemma.

Lemma 2.2. $L^*_{\mathcal{M}} \geq \sum\limits_{j=1}^{K} \frac{a^j_{\mathcal{M}}}{N} \cdot \frac{K-j}{Kj}$.

Next, we first demonstrate the converse of Theorem 2.1 using Lemma 2.2, and then give the proof of Lemma 2.2.

Converse Proof of Theorem 2.1. It is clear that the minimum communication load $L^*(r)$ is lower bounded by the minimum value of $L^*_{\mathcal{M}}$ over all possible file assignments which admit a computation load of r:

$$L^*(r) \geq \inf_{\mathcal{M}:|\mathcal{M}_1|+\cdots+|\mathcal{M}_K|=rN} L^*_{\mathcal{M}}. \tag{2.12}$$

Then by Lemma 2.2, we have

$$L^*(r) \geq \inf_{\mathcal{M}:|\mathcal{M}_1|+\cdots+|\mathcal{M}_K|=rN} \sum_{j=1}^{K} \frac{a^j_{\mathcal{M}}}{N} \cdot \frac{K-j}{Kj}. \tag{2.13}$$

For every file assignment $\mathcal{M}$ such that $|\mathcal{M}_1| + \cdots + |\mathcal{M}_K| = rN$, $\{a^j_{\mathcal{M}}\}_{j=1}^K$ satisfy

$$a^j_{\mathcal{M}} \geq 0,\ j \in \{1,...,K\}, \tag{2.14}$$

$$\sum_{j=1}^{K} a^j_{\mathcal{M}} = N, \tag{2.15}$$

$$\sum_{j=1}^{K} j a^j_{\mathcal{M}} = rN. \tag{2.16}$$

Then since the function $\frac{K-j}{Kj}$ in (2.13) is convex in j, and by (2.15) $\sum\limits_{j=1}^{K} \frac{a_{\mathcal{M}}^j}{N} = 1$, (2.13) becomes

$$L^*(r) \geq \inf_{\mathcal{M}:|\mathcal{M}_1|+\cdots+|\mathcal{M}_K|=rN} \frac{K - \sum\limits_{j=1}^{K} j\frac{a_{\mathcal{M}}^j}{N}}{K\sum\limits_{j=1}^{K} j\frac{a_{\mathcal{M}}^j}{N}} \overset{(a)}{=} \frac{K-r}{Kr}, \tag{2.17}$$

where (a) is due to the requirement imposed by the computation load in (2.16).

The lower bound on $L^*(r)$ in (2.17) holds for general $1 \leq r \leq K$. We can further improve the lower bound for non-integer valued r as follows. For a particular $r \notin \mathbb{N}$, we first find the line $p+qj$ as a function of $1 \leq j \leq K$ connecting the two points $(\lfloor r \rfloor, \frac{K-\lfloor r \rfloor}{K\lfloor r \rfloor})$ and $(\lceil r \rceil, \frac{K-\lceil r \rceil}{K\lceil r \rceil})$. More specifically, we find $p, q \in \mathbb{R}$ such that

$$p + qj|_{j=\lfloor r \rfloor} = \frac{K-\lfloor r \rfloor}{K\lfloor r \rfloor}, \tag{2.18}$$

$$p + qj|_{j=\lceil r \rceil} = \frac{K-\lceil r \rceil}{K\lceil r \rceil}. \tag{2.19}$$

Then by the convexity of the function $\frac{K-j}{Kj}$ in j, we have for integer-valued $j = 1, \ldots, K$,

$$\frac{K-j}{Kj} \geq p + qj, \quad j = 1, \ldots, K. \tag{2.20}$$

Then (2.13) reduces to

$$L^*(r) \geq \inf_{\mathcal{M}:|\mathcal{M}_1|+\cdots+|\mathcal{M}_K|=rN} \sum_{j=1}^{K} \frac{a_{\mathcal{M}}^j}{N} \cdot (p+qj) \tag{2.21}$$

$$= \inf_{\mathcal{M}:|\mathcal{M}_1|+\cdots+|\mathcal{M}_K|=rN} \sum_{j=1}^{K} \frac{a_{\mathcal{M}}^j}{N} \cdot p + \sum_{j=1}^{K} \frac{j a_{\mathcal{M}}^j}{N} \cdot q \tag{2.22}$$

$$\overset{(b)}{=} p + qr, \tag{2.23}$$

where (b) is due to the constraints on $\{a_{\mathcal{M}}^j\}_{j=1}^{K}$ in (2.15) and (2.16).

Therefore, $L^*(r)$ is lower bounded by the lower convex envelop of the points $\{(r, \frac{K-r}{Kr}) : r \in \{1, ..., K\}\}$. This completes the proof of the converse part of Theorem 2.1. ■

We devote the rest of this subsection to the proof of Lemma 2.2. To prove Lemma 2.2, we develop a lower bound on the number of bits communicated by any subset of nodes, by induction on the size of the subset.

Proof of Lemma 2.2. For $q \in \{1, ..., Q\}$, $n \in \{1, ..., N\}$, we let $V_{q,n}$ be i.i.d. random variables uniformly distributed on $\mathbb{F}_{2^T}$. We let the intermediate values $v_{q,n}$ be the realizations of $V_{q,n}$. For some $\mathcal{Q} \subseteq \{1, \ldots, Q\}$ and $\mathcal{N} \subseteq \{1, \ldots, N\}$, we define

$$V_{\mathcal{Q},\mathcal{N}} \triangleq \{V_{q,n} : q \in \mathcal{Q}, n \in \mathcal{N}\}. \tag{2.24}$$

Since each message X_k is generated as a function of the intermediate values that are computed at Node k, we have for all $k \in \{1, ..., K\}$,

$$H(X_k | V_{:,\mathcal{M}_k}) = 0, \tag{2.25}$$

where we use ":" to denote the set of all possible indices.

The validity of the shuffling scheme requires that for all $k \in \{1, ..., K\}$, the following equation holds:

$$H(V_{\mathcal{W}_k,:} | X_:, V_{:,\mathcal{M}_k}) = 0. \tag{2.26}$$

For a subset $\mathcal{S} \subseteq \{1, ..., K\}$, we define

$$Y_{\mathcal{S}} \triangleq (V_{\mathcal{W}_{\mathcal{S}},:}, V_{:,\mathcal{M}_{\mathcal{S}}}), \tag{2.27}$$

which contains all the intermediate values required by the nodes in $\mathcal{S}$ and all the intermediate values known locally by the nodes in $\mathcal{S}$ after the Map phase.

For any subset $\mathcal{S} \subseteq \{1, \ldots, K\}$ and a file assignment $\mathcal{M}$, we denote the number of files that are *exclusively* mapped by j nodes in $\mathcal{S}$ as $a_{\mathcal{M}}^{j,\mathcal{S}}$:

$$a_{\mathcal{M}}^{j,\mathcal{S}} \triangleq \sum_{\mathcal{J} \subseteq \mathcal{S} : |\mathcal{J}| = j} |(\underset{k \in \mathcal{J}}{\cap} \mathcal{M}_k) \backslash (\underset{i \notin \mathcal{J}}{\cup} \mathcal{M}_i)|, \tag{2.28}$$

and the message symbols communicated by the nodes whose indices are in $\mathcal{S}$ as

$$X_{\mathcal{S}} \triangleq \{X_k : k \in \mathcal{S}\}. \tag{2.29}$$

Then we prove the following claim.

Claim 2.2.1. For any subset $\mathcal{S} \subseteq \{1, ..., K\}$, we have

$$H(X_{\mathcal{S}}|Y_{\mathcal{S}^c}) \geq T \sum_{j=1}^{|\mathcal{S}|} a_{\mathcal{M}}^{j,\mathcal{S}} \frac{Q}{K} \cdot \frac{|\mathcal{S}| - j}{j}, \tag{2.30}$$

where $\mathcal{S}^c \triangleq \{1, \ldots, K\} \backslash \mathcal{S}$ denotes the complement of $\mathcal{S}$. □

We prove Claim 2.2.1 by induction.

a. If $\mathcal{S} = \{k\}$ for any $k \in \{1, \ldots, K\}$, obviously

$$H(X_k|Y_{\{1,\ldots,K\}\backslash\{k\}}) \geq 0 = T a_{\mathcal{M}}^{1,\{k\}} \frac{Q}{K} \cdot \frac{1-1}{1}. \tag{2.31}$$

b. Suppose the statement is true for all subsets of size S_0.

For any $\mathcal{S} \subseteq \{1, ..., K\}$ of size $|\mathcal{S}| = S_0 + 1$ and any $k \in \mathcal{S}$, we have

$$\begin{aligned} H(X_{\mathcal{S}}|Y_{\mathcal{S}^c}) &= \frac{1}{|\mathcal{S}|} \sum_{k \in \mathcal{S}} H(X_{\mathcal{S}}, X_k|Y_{\mathcal{S}^c}) && (2.32) \\ &\geq \frac{1}{|\mathcal{S}|} \sum_{k \in \mathcal{S}} H(X_{\mathcal{S}}|X_k, Y_{\mathcal{S}^c}) + \frac{1}{|\mathcal{S}|} H(X_{\mathcal{S}}|Y_{\mathcal{S}^c}). && (2.33) \end{aligned}$$

From (2.33), we have

$$\begin{aligned} H(X_{\mathcal{S}}|Y_{\mathcal{S}^c}) &\geq \frac{1}{|\mathcal{S}| - 1} \sum_{k \in \mathcal{S}} H(X_{\mathcal{S}}|X_k, Y_{\mathcal{S}^c}) && (2.34) \\ &\geq \frac{1}{S_0} \sum_{k \in \mathcal{S}} H(X_{\mathcal{S}}|X_k, V_{:,\mathcal{M}_k}, Y_{\mathcal{S}^c}) && (2.35) \\ &= \frac{1}{S_0} \sum_{k \in \mathcal{S}} H(X_{\mathcal{S}}|V_{:,\mathcal{M}_k}, Y_{\mathcal{S}^c}). && (2.36) \end{aligned}$$

Due to the decodability criterion at Node k, for each $k \in \mathcal{S}$, the term on the RHS of (2.36) can be written as

$$\begin{aligned} H(X_{\mathcal{S}}|V_{:,\mathcal{M}_k}, Y_{\mathcal{S}^c}) &= H(X_{\mathcal{S}}, V_{\mathcal{W}_k,:}|V_{:,\mathcal{M}_k}, Y_{\mathcal{S}^c}) && (2.37) \\ &= H(V_{\mathcal{W}_k,:}|V_{:,\mathcal{M}_k}, Y_{\mathcal{S}^c}) + H(X_{\mathcal{S}}|V_{\mathcal{W}_k,:}, V_{:,\mathcal{M}_k}, Y_{\mathcal{S}^c}). && (2.38) \end{aligned}$$

The first term on the RHS of (2.38) can be lower bounded as follows.

$$H(V_{\mathcal{W}_k,:}|V_{:,\mathcal{M}_k},Y_{\mathcal{S}^c}) = H(V_{\mathcal{W}_k,:}|V_{:,\mathcal{M}_k},V_{\mathcal{W}_{\mathcal{S}^c},:},V_{:,\mathcal{M}_{\mathcal{S}^c}}) \tag{2.39}$$
$$\overset{(a)}{=} H(V_{\mathcal{W}_k,:}|V_{:,\mathcal{M}_k},V_{:,\mathcal{M}_{\mathcal{S}^c}}) \tag{2.40}$$
$$\overset{(b)}{=} \sum_{q\in\mathcal{W}_k} H(V_{\{q\},:}|V_{\{q\},\mathcal{M}_k\cup\mathcal{M}_{\mathcal{S}^c}}) \tag{2.41}$$
$$\overset{(c)}{=} \frac{Q}{K}T\sum_{j=0}^{S_0} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}} \geq \frac{Q}{K}T\sum_{j=1}^{S_0} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}}, \tag{2.42}$$

where (a) is due to the independence of intermediate values and the fact that $\mathcal{W}_k \cap \mathcal{W}_{\mathcal{S}^c} = \varnothing$ (different nodes calculate different output functions), (b) is due to the independence of intermediate values, and (c) is due to the independence of the intermediate values and the fact that $|\mathcal{W}_k| = \frac{Q}{K}$.

The second term on the RHS of (2.38) can be lower bounded by the induction assumption:

$$H(X_{\mathcal{S}}|V_{\mathcal{W}_k,:},V_{:,\mathcal{M}_k},Y_{\mathcal{S}^c}) = H(X_{\mathcal{S}\backslash\{k\}}|Y_{(\mathcal{S}\backslash\{k\})^c}) \tag{2.43}$$
$$\geq T\sum_{j=1}^{S_0} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}}\frac{Q}{K}\cdot\frac{S_0-j}{j}. \tag{2.44}$$

Thus by (2.36), (2.38), (2.42) and (2.44), we have

$$H(X_{\mathcal{S}}|Y_{\mathcal{S}^c}) \geq \frac{1}{S_0}\sum_{k\in\mathcal{S}}\left(T\sum_{j=1}^{S_0} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}}\frac{Q}{K} + T\sum_{j=1}^{S_0} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}}\frac{Q}{K}\cdot\frac{S_0-j}{j}\right) \tag{2.45}$$
$$= \frac{T}{S_0}\sum_{k\in\mathcal{S}}\sum_{j=1}^{S_0} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}}\frac{Q}{K}\cdot\frac{S_0}{j} = T\sum_{j=1}^{S_0}\frac{Q}{K}\cdot\frac{1}{j}\sum_{k\in\mathcal{S}} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}}. \tag{2.46}$$

By the definition of $a_{\mathcal{M}}^{j,\mathcal{S}}$, we have the following equations.

$$\begin{aligned}\sum_{k\in\mathcal{S}} a_{\mathcal{M}}^{j,\mathcal{S}\backslash\{k\}} =&\sum_{k\in\mathcal{S}}\sum_{n=1}^{N} \mathbb{1}(\text{file } n \text{ is only mapped by some nodes}\\ &\text{in } \mathcal{S}\backslash\{k\})\cdot\mathbb{1}(\text{file } n \text{ is mapped by } j \text{ nodes}) \qquad (2.47)\\ =&\sum_{n=1}^{N}\mathbb{1}(\text{file } n \text{ is only mapped by } j \text{ nodes in } \mathcal{S})\cdot\sum_{k\in\mathcal{S}}\mathbb{1}(\text{file}\\ &n \text{ is not mapped by Node } k) \qquad (2.48)\\ =&\sum_{n=1}^{N}\mathbb{1}(\text{file } n \text{ is only mapped by } j \text{ nodes in } \mathcal{S})(|\mathcal{S}|-j)\\ & \qquad (2.49)\\ =&a_{\mathcal{M}}^{j,\mathcal{S}}(S_0+1-j). \qquad (2.50)\end{aligned}$$

Applying (2.50) to (2.46) yields

$$H(X_{\mathcal{S}}|Y_{\mathcal{S}^c}) \geq T\sum_{j=1}^{S_0+1} a_{\mathcal{M}}^{j,\mathcal{S}}\frac{Q}{K}\cdot\frac{S_0+1-j}{j}. \qquad (2.51)$$

c. Thus for all subsets $\mathcal{S}\subseteq\{1,...,K\}$, the following equation holds:

$$H(X_{\mathcal{S}}|Y_{\mathcal{S}^c}) \geq T\sum_{j=1}^{|\mathcal{S}|} a_{\mathcal{M}}^{j,\mathcal{S}}\frac{Q}{K}\cdot\frac{|\mathcal{S}|-j}{j}, \qquad (2.52)$$

which proves Claim 2.2.1.

Then by Claim 2.2.1, let $\mathcal{S}=\{1,...,K\}$ be the set of all K nodes,

$$L_{\mathcal{M}}^{*} \geq \frac{H(X_{\mathcal{S}}|Y_{\mathcal{S}^c})}{QNT} \geq \sum_{j=1}^{K}\frac{a_{\mathcal{M}}^{j}}{N}\cdot\frac{K-j}{Kj}. \qquad (2.53)$$

This completes the proof of Lemma 2.2. ■

2.2 Empirical evaluations of coded distributed computing

In this section, we apply the Coded Distributed Computing (CDC) scheme proposed in the previous section to a widely-used distributed sorting algorithm, `TeraSort` *Hadoop TeraSort* n.d., developing a coded distributed sorting algorithm `CodedTeraSort`. While the run-time performance of `TeraSort` is known to be severely limited by the data shuffling time between distributed computing nodes (see, e.g., Guo *et al.*, 2013; Zhang *et al.*, 2013), `CodedTeraSort` injects and leverages extra local computations to trade for a substantially smaller bandwidth consumption, hence significantly improving the overall run-time performance over `TeraSort`.

2.2.1 Execution time of Coded Distributed Computing

For a MapReduce application whose overall response time is composed of the time spent executing the Map tasks, denoted by T_{map}, the time spent shuffling intermediate values, denoted by T_{shuffle}, and the time spent executing the Reduce tasks, denoted by T_{reduce}, we have

$$T_{\text{total, MR}} = T_{\text{map}} + T_{\text{shuffle}} + T_{\text{reduce}}. \tag{2.54}$$

Using CDC, we can leverage $r\times$ more computations in the Map phase, in order to reduce the communication load by the same multiplicative factor, where $r \in \mathbb{N}$ is a design parameter that can be optimized to minimize the overall execution time. Hence, CDC promises that we can achieve the overall execution time of

$$T_{\text{total, CDC}} \approx rT_{\text{map}} + \tfrac{1}{r}T_{\text{shuffle}} + T_{\text{reduce}}, \tag{2.55}$$

for any $1 \leq r \leq K$, where K is the total number of nodes on which the distributed computation is executed. To minimize the above execution time, one would choose

$$r^* = \left\lfloor \sqrt{\tfrac{T_{\text{shuffle}}}{T_{\text{map}}}} \right\rfloor \text{ or } \left\lceil \sqrt{\tfrac{T_{\text{shuffle}}}{T_{\text{map}}}} \right\rceil,$$

resulting in execution time of

$$T^*_{\text{total, CDC}} \approx 2\sqrt{T_{\text{shuffle}}T_{\text{map}}} + T_{\text{reduce}}. \tag{2.56}$$

2.2.2 TeraSort

`TeraSort` O'Malley, 2008 is a conventional algorithm for distributed sorting of a large amount of data. The input data to be sorted is in the format of key-value (KV) pairs, meaning each input KV pair consists of a key and a value. For example, the domain of the keys can be 10-byte integers, and the domain of the values can be arbitrary strings. `TeraSort` aims to sort the input data according to their keys, e.g., sorting integers. A `TeraSort` algorithm run over K nodes, whose indices are denoted by a set $\mathcal{K} = \{1, \ldots, K\}$, is comprised of the following five components.

File placement. Let F denote the entire KV pairs to be sorted. They are split into K disjoint input files, denoted by $F_{\{1\}}, \ldots, F_{\{K\}}$. File $F_{\{k\}}$ is assigned to and locally stored at Node k.

Key domain partitioning. The key domain of the KV pair, denoted by P, is split into K *ordered* partitions, denoted by $P_1, \ldots, P_K$. Specifically, for any $p \in P_i$ and any $p' \in P_{i+1}$, it holds that $p < p'$ for all $i \in \{1, \ldots, K-1\}$. For example, when $P = [0, 100]$ and $K = 4$, the partitions can be $P_1 = [0, 25), P_2 = [25, 50), P_3 = [50, 75), P_4 = [75, 100]$. Node k is responsible for sorting all KV pairs in the partition P_k, for all $k \in \mathcal{K}$.

Map stage. Each node hashes each KV pair in the locally stored file $F_{\{k\}}$ to the partition its key falls into. For each of the K key partitions, the hashing procedure on the file $F_{\{k\}}$ generates an *intermediate value* that contains the KV pairs in $F_{\{k\}}$ whose keys belong to that partition.

More specifically, we denote the intermediate value of the partition P_j from the file $F_{\{k\}}$ as $I^j_{\{k\}}$, and the hashing procedure on the file $F_{\{k\}}$ is defined as

$$\left\{I^1_{\{k\}}, \ldots, I^K_{\{k\}}\right\} \leftarrow \mathit{Hash}\Big(F_{\{k\}}\Big).$$

Shuffle stage. The intermediate value $I^k_{\{j\}}$ calculated at Node j, $j \neq k$, is unicast to Node k from Node j, for all $k \in \mathcal{K}$. Since the intermediate value $I^k_{\{k\}}$ is computed locally at Node k in the Map stage, by the end of the Shuffle stage, Node k knows all intermediate values $\left\{I^k_{\{1\}}, \ldots, I^k_{\{K\}}\right\}$ of the partition P_k from all K files.

Reduce stage. Node k locally sorts all KV pairs whose keys fall into the partition P_k, for all $k \in \mathcal{K}$. Specifically, it sorts all intermediate values in the partition P_k into a sorted list Q_k as follows

$$Q_k \leftarrow \mathit{Sort}\Big(\left\{I^k_{\{1\}}, \ldots, I^k_{\{K\}}\right\}\Big).$$

Table 2.1: Performance of `TeraSort` sorting 12GB data with $K = 16$ nodes and 100 Mbps network speed

Map (sec.)	Pack (sec.)	Shuffle (sec.)	Unpack (sec.)	Reduce (sec.)	Total (sec.)
1.86	2.35	945.72	0.85	10.47	961.25

Performance evaluation

We performed an experiment on Amazon EC2 to sort 12GB of data by running `TeraSort` on 16 nodes. The breakdown of the total execution time is shown in Table 2.1.

We observe from Table 2.1 that for a conventional `TeraSort` execution, 98.4% of the total execution time was spent in data shuffling, which is 508.5× of the time spent in the Map stage. This motivates us to develop a coded distributed sorting algorithm, named `CodedTeraSort`, which integrates the coding technique of CDC into `TeraSort` to trade extra computation time to significantly reduce the communication time, as shown in (2.55).

2.2.3 Coded TeraSort

We describe the `CodedTeraSort` algorithm, which is developed by integrating the coding techniques of the CDC into the `TeraSort` algorithm.
Structured redundant file placement. For some parameter $r \in \{1, \ldots, K\}$, we first split the entire input KV pairs into $N = \binom{K}{r}$ input files. Unlike the file placement of `TeraSort`, `CodedTeraSort` places each of the N input files *repetitively* on r distinct nodes.

We label an input file using a unique subset $\mathcal{S}$ of $\mathcal{K}$ with size $|\mathcal{S}| = r$, i.e., the N input files are denoted by

$$\{F_{\mathcal{S}} : \mathcal{S} \subseteq \mathcal{K}, |\mathcal{S}| = r\}. \tag{2.57}$$

We repetitively place an input file $F_{\mathcal{S}}$ on each of the r nodes in $\mathcal{S}$, and hence each node now stores $Nr/K = \binom{K-1}{r-1}$ files. As illustrated in a simple example in Figure 2.5 for $K = 4$ and $r = 2$, the file $F_{\{2,3\}}$ is placed on Nodes 2 and 3. Node 2 has files $F_{\{1,2\}}, F_{\{2,3\}}, F_{\{2,4\}}$.

As is done in the `TeraSort`, the key domain of the input KV pairs is split into K ordered partitions $P_1, \ldots, P_K$, and Node k is responsible for sorting all KV pairs in the partition P_k in the Reduce stage, for all $k \in \mathcal{K}$.

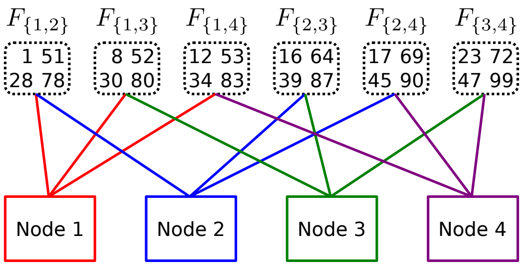


Figure 2.5: An illustration of the structured redundant file placement in `CodedTeraSort` with $K = 4$ nodes and $r = 2$.

Map stage. Each node repeatedly performs the Map stage operation of `TeraSort` described above, on each input file placed on that node. Only relevant intermediate values generated in the Map stage are kept locally for further processing. In particular, out of the K intermediate values $\left\{I^1_{\mathcal{S}}, \ldots, I^K_{\mathcal{S}}\right\}$ generated from file $F_{\mathcal{S}}$, only $I^k_{\mathcal{S}}$ and $\{I^i_{\mathcal{S}} : i \in \mathcal{K} \backslash \mathcal{S}\}$ are kept at Node k. This is because that the intermediate value $I^i_{\mathcal{S}}$, required by Node $i \in \mathcal{S} \backslash \{k\}$ in the Reduce stage, is already available at Node i after the Map stage, so Node k does not need to keep them and send them to the nodes in $\mathcal{S} \backslash \{k\}$. For example, as shown in Figure 2.6, Node 1 does not keep the intermediate value $I^2_{\{1,2\}}$ for Node 2. However, Node 1 keeps $I^1_{\{1,2\}}, I^3_{\{1,2\}}, I^4_{\{1,2\}}$, which are required by Nodes 1, 3, and 4 in the Reduce stage.

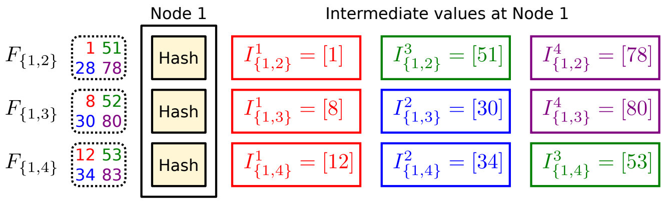


Figure 2.6: An illustration of the Map stage at Node 1 in `CodedTeraSort` with $K = 4$, $r = 2$ and the key partitions $[0, 25), [25, 50), [50, 75), [75, 100]$.

Encoding to create coded packets. The role of the encoding process is to exploit the structured data redundancy to create coded multicast packets that are simultaneously useful for multiple nodes, thus saving the load of communicating intermediate values. Specifically, in every subset $\mathcal{M} \subseteq \mathcal{K}$ of $|\mathcal{M}| = r+1$ nodes, the encoding operation proceeds as follows.

- For each $t \in \mathcal{M}$, the intermediate value $I^t_{\mathcal{M}\backslash\{t\}}$, which is know at all nodes in $\mathcal{M}\backslash\{t\}$, is evenly and arbitrarily split into r segments, i.e.,

$$I^t_{\mathcal{M}\backslash\{t\}} = \{I^t_{\mathcal{M}\backslash\{t\},k} : k \in \mathcal{M}\backslash\{t\}\}, \tag{2.58}$$

  where $I^t_{\mathcal{M}\backslash\{t\},k}$ denotes the segment corresponding to Node k.

- For each $k \in \mathcal{M}$, we generate the coded packet of Node k in $\mathcal{M}$, denoted by $E_{\mathcal{M},k}$, by XORing all segments corresponding to Node k in $\mathcal{M}$, [4] i.e.,

$$E_{\mathcal{M},k} = \bigoplus_{t\in\mathcal{M}\backslash\{k\}} I^t_{\mathcal{M}\backslash\{t\},k}. \tag{2.59}$$

By the end of the Encoding stage, for each $k \in \mathcal{K}$, Node k has generated $\binom{K-1}{r}$ coded packets, i.e., $\{E_{\mathcal{M},k} : k \in \mathcal{M}, |\mathcal{M}| = r+1\}$.

In Figure 2.7, we consider a scenario with $r = 2$, and illustrate the encoding process in the subset $\mathcal{M} = \{1,2,3\}$. Exploiting the particular structure imposed in the stage of file placement, each node creates a coded packet that contains data segments useful for the other 2 nodes.

Multicast shuffling. After all coded packets are created at the K nodes, the multicast shuffling process takes place within each subset of $r+1$ nodes. Specifically, within each group $\mathcal{M} \subseteq \mathcal{K}$ of $|\mathcal{M}| = r+1$ nodes, each Node $k \in \mathcal{M}$ multicasts its coded packet $E_{\mathcal{M},k}$ to the other nodes in $\mathcal{M}\backslash\{k\}$. This coded packet is simultaneously useful for all of these r nodes.

Decoding. Having received the coded packet $E_{\mathcal{M},u}$ from Node u, Node $k \in \mathcal{M}\backslash\{u\}$ performs the decoding process by XORing the data segments $\{I^t_{\mathcal{M}\backslash\{t\},u} : t \in \mathcal{M}\backslash\{u,k\}\}$ with $E_{\mathcal{M},u}$ to recover the

[4]All segments are zero-padded to the length of the longest one.

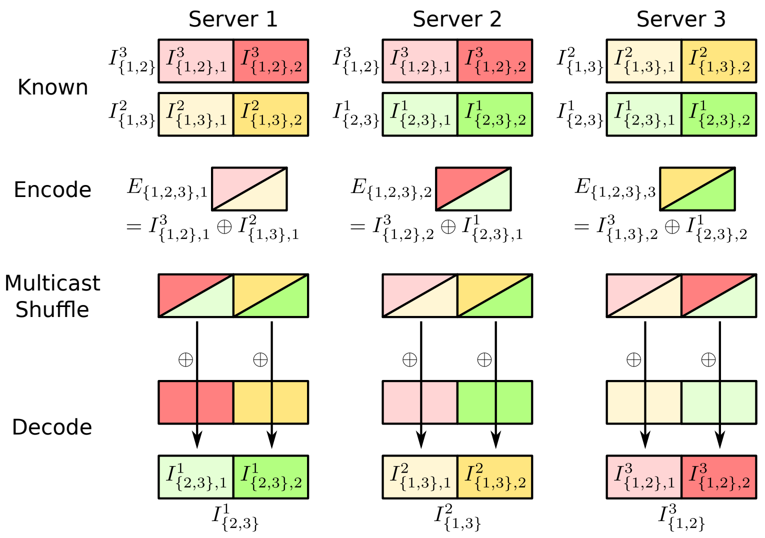


Figure 2.7: An illustration of the encoding process within a multicast group $\mathcal{M} = \{1, 2, 3\}$.

desired segment $I^k_{\mathcal{M}\backslash\{k\},u}$. Similarly, Node k recovers all data segments $\{I^k_{\mathcal{M}\backslash\{k\},u} : u \in \mathcal{M}\backslash\{k\}\}$ from the received coded packets in $\mathcal{M}$, and merge them back to obtain a required intermediate value $I^k_{\mathcal{M}\backslash\{k\}}$.
Reduce. After the Decoding stage, Node k has obtained all KV pairs in the partition P_k, for all $k \in \mathcal{K}$. In this final stage, Node k, $k = 1, \ldots, K$, performs the Reduce process as in the `TeraSort` algorithm, sorting the KV pairs in partition P_k locally.

2.2.4 Experiments

We imperially demonstrate the performance gain of `CodedTeraSort` through experiments on Amazon EC2 clusters. In this subsection, we first present the choices we have made for the implementation. Then, we describe experiment setup. Finally, we discuss the experiment results.

Implementation Choices

Data format. All input KV pairs are generated from `TeraGen` *Hadoop TeraSort* n.d. in the standard Hadoop package. Each input KV pair consists of a 10-byte key and a 90-byte value. A key is a 10-byte unsigned integer, and the value is an arbitrary string of 90 bytes. The KV pairs are sorted based on their keys, using the standard integer ordering.

Platform and library. We choose Amazon EC2 as the evaluation platform. We implement both `TeraSort` and `CodedTeraSort` algorithms in `C++`, and use Open MPI library *Open MPI: Open Source High Performance Computing* n.d. for communications among EC2 instances.

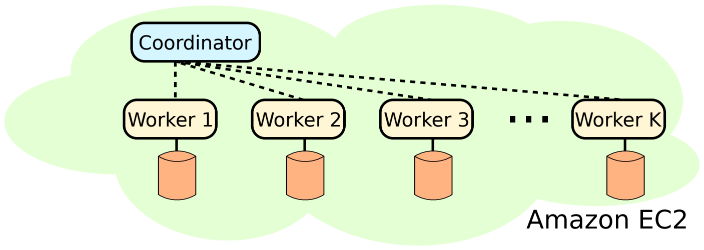


Figure 2.8: The coordinator-worker system architecture.

System architecture. As shown in Figure 2.8, we employ a system architecture that consists of a coordinator node and K worker nodes, for some $K \in \mathbb{N}$. Each node is run as an EC2 instance. The coordinator node is responsible for creating the key partitions and placing the input files on the local disks of the worker nodes. The worker nodes are responsible for distributedly executing the stages of the sorting algorithms.

In-memory processing. After the KV pairs are loaded from the local files into the workers' memories, all intermediate data that are used for encoding, decoding and local sorting are persisted in the memories, and hence there is no disk I/O involved during the executions of the algorithms.

In the `TeraSort` implementation, each node sequentially steps through Map, Pack, Shuffle, Unpack, and Reduce stages. In the Reduce stage, the standard sort `std::sort` is used to sort each partition locally. To better interpret the experiment results, we add the Pack and the Unpack stages to separate the time of serialization and deserialization from the other stages. The Pack stage serializes each intermediate value to a continuous memory array to ensure that a single TCP flow is

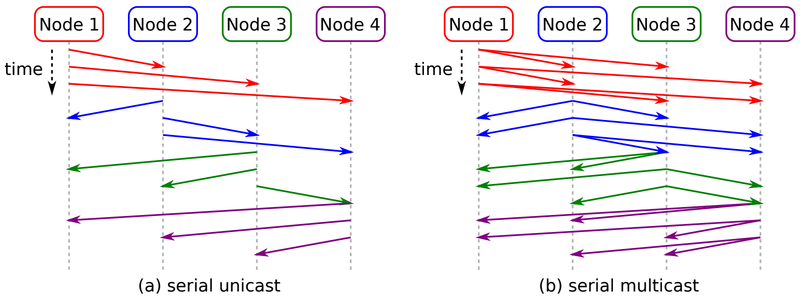


Figure 2.9: (a) Serial unicast in the Shuffle stage of `TeraSort`; a solid arrow represents a unicast. (b) Serial multicast in the Multicast Shuffle stage of `CodedTeraSort`; a group of solid arrows starting at the same node represents a multicast.

created for each intermediate value (which may contain multiple KV pairs) when `MPI_Send` is called[5]. The Unpack stage deserializes the received data to a list of KV pairs. In the Shuffle stage, intermediate values are unicast serially, meaning that there is only one sender node and one receiver node at any time instance. Specifically, as illustrated in Figure 2.9(a), Node 1 starts to unicast to Nodes 2, 3, and 4 back-to-back. After Node 1 finishes, Node 2 unicasts back-to-back to Nodes 1, 3, and 4. This continues until Node 4 finishes.

In the `CodedTeraSort` implementation, each node sequentially steps through CodeGen, Map, Encode, Multicast Shuffling, Decode, and Reduce stages. In the CodeGen (or code generation) stage, firstly, each node generates all file indices, as subsets of r nodes. Then each node uses `MPI_Comm_split` to initialize $\binom{K}{r+1}$ multicast groups each containing $r+1$ nodes on Open MPI, such that multicast communications will be performed within each of these groups. The serialization and deserialization are implemented respectively in the Encode and the Decode stages. In Multicast Shuffling, `MPI_Bcast` is called to multicast a coded packet in a serial manner, so only one node multicasts one of its encoded packets at any time instance. Specifically, as illustrated in Figure 2.9(b), Node 1 multicasts to the other 2 nodes in each multicast group Node 1 is in. For example, Node 1 first multicasts to Node 2 and 3

[5]Creating a TCP flow per KV pair leads to inefficiency from overhead and convergence issue.

Table 2.2: Sorting 12 GB data with $K = 16$ nodes and 100 Mbps network speed

	CodeGen (sec.)	Map (sec.)	Pack/Encode (sec.)	Shuffle (sec.)	Unpack/Decode (sec.)	Reduce (sec.)	Total Time (sec.)	Speedup
`TeraSort`:	–	1.86	2.35	945.72	0.85	10.47	961.25	
`CodedTeraSort`: $r = 3$	6.06	6.03	5.79	412.22	2.41	13.05	445.56	2.16×
`CodedTeraSort`: $r = 5$	23.47	10.84	8.10	222.83	3.69	14.40	283.33	3.39×

Table 2.3: Sorting 12 GB data with $K = 20$ nodes and 100 Mbps network speed

	CodeGen (sec.)	Map (sec.)	Pack/Encode (sec.)	Shuffle (sec.)	Unpack/Decode (sec.)	Reduce (sec.)	Total Time (sec.)	Speedup
`TeraSort`:	–	1.47	2.00	960.07	0.62	8.29	972.45	
`CodedTeraSort`: $r = 3$	19.32	4.68	4.89	453.37	1.87	9.73	493.86	1.97×
`CodedTeraSort`: $r = 5$	140.91	8.59	7.51	269.42	3.70	10.97	441.10	2.20×

in the group $\{1, 2, 3\}$. After Node 1 finishes, Node 2 starts multicasting in the same manner. This process continues until Node 4 finishes.

Experiment Setup

We conduct experiments using the following configurations to evaluate the performance of `CodedTeraSort` and `TeraSort` on Amazon EC2:

- The coordinator runs on a r3.large instance with 2 processors, 15 GB memory, and 32 GB SSD.
- Each worker node runs on an m3.large instance with 2 processors, 7.5 GB memory, and 32 GB SSD.
- The incoming and outgoing traffic rates of each instance are limited to 100 Mbps.[6]
- 12 GB of input data (equivalently 120 M KV pairs) is sorted.

2.2.5 Results

The breakdowns of the execution times with $K = 16$ workers and $K = 20$ workers are shown in Tables 2.2 and 2.3 respectively. We observe an overall 1.97×-3.39× speedup of `CodedTeraSort` as compared

[6]This is to alleviate the effects of the bursty behaviors of the transmission rates in the beginning of some TCP sessions. The rates are limited by traffic control command `tc` *tc - show / manipulate traffic control settings* n.d.

with `TeraSort`. From the experiment results we make the following observations:

- For `CodedTeraSort`, the time spent in the CodeGen stage is proportional to $\binom{K}{r+1}$, which is the number of multicast groups.
- The Map time of `CodedTeraSort` is approximately r times higher than that of `TeraSort`. This is because that each node hashes r times more KV pairs than that in `TeraSort`. Specifically, the ratios of the `CodedTeraSort`'s Map time to the `TeraSort`'s Map time from Table 2.2 are $6.03/1.86 \approx 3.2$ and $10.84/1.86 \approx 5.8$, and from Table 2.3 are $4.68/1.47 \approx 3.2$ and $8.59/1.47 \approx 5.8$.
- While `CodedTeraSort` theoretically promises a factor of more than $r\times$ reduction in shuffling time, the actual gains observed in the experiments are slightly less than r. For example, for an experiment with $K = 16$ nodes and $r = 3$, as shown in Table 2.2, the speedup of the Shuffle stage is $945.72/412.22 \approx 2.3 < 3$. This phenomenon is caused by the following two factors. 1) Open MPI's multicast API (`MPI_Bcast`) has an inherent overhead per multicast group, for instance, a multicast tree is constructed before multicasting to a set of nodes. 2) Using the `MPI_Bcast` API, the time of multicasting a packet to r nodes is higher than that of unicasting the same packet to a single node. In fact, as measured in Lee *et al.*, 2018, the multicasting time increases logarithmically with r.
- The sorting times in the Reduce stage of both algorithms depend on the available memories of the nodes. `CodedTeraSort` inherently has a higher memory overhead, e.g., it requires persisting more intermediate values in the memories than `TeraSort` for coding purposes, hence its local sorting process takes slightly longer. This can be observed from the Reduce column in Tables 2.2 and 2.3.

Further, we observe the following trends from both tables:

Impact of redundancy parameter r: As r increases, the shuffling time reduces substantially by approximately r times. However, the Map execution time increases linearly with r, and more importantly the

CodeGen time increases as $\binom{K}{r+1}$. Hence, for small values of r ($r < 6$) we observe overall reduction in execution time, and the speedup increases. However, as we further increase r, the CodeGen time will dominate the execution time, and the speedup decreases. Hence, in our evaluations, we have limited r to be at most 5.[7]

Impact of worker number K: As K increases, the speedup decreases. This is due to the following two reasons. 1) The number of multicast groups, i.e., $\binom{K}{r+1}$, grows exponentially with K, resulting in a longer execution time of the CodeGen process. 2) When more nodes participate in the computation, for a fixed r, less amount of KV pairs are hashed at each node locally in the Map stage, resulting in less locally available intermediate values and a higher communication load.

2.3 Extension to wireless distributed computing

Having theoretically and empirically demonstrated how coding can help to overcome the communication bottlenecks and significantly improve the performance of applications hosted over wireline networks like datacenters, we also extend the idea of coded computing into mobile edge computing, in which mobile users participating in the computation exchange intermediate computation results via the underlying wireless links. In the mobile edge computing scenario, the communication bottleneck becomes much worse due to much lower data rates of wireless networks, which significantly delays the overall computation. We demonstrate in this section that coding can exploit the rather abundant computation resources in the network to create redundant computations, and trade these redundant computations for substantial reduction on the bandwidth requirement. This technology will enable a *scalable* mobile computing platform that can accommodates a large number of users with a fixed communication bandwidth.

[7]The redundancy parameter r is also limited by the total storage available at the nodes. Since for a choice of redundancy parameter r, each piece of input KV pairs should be stored at r nodes, we can not increase r beyond $\frac{\text{total available storage at the worker nodes}}{\text{input size}}$.

2.3.1 System model

We consider a system that has K mobile users. As illustrated in Figure 2.10, all users are connected wirelessly to an access point (e.g., a cellular base station or a Wi-Fi router). The uplink channels of the K users towards the access point are orthogonal to each other, and the signals transmitted by the access point on the downlink are received by all the users.

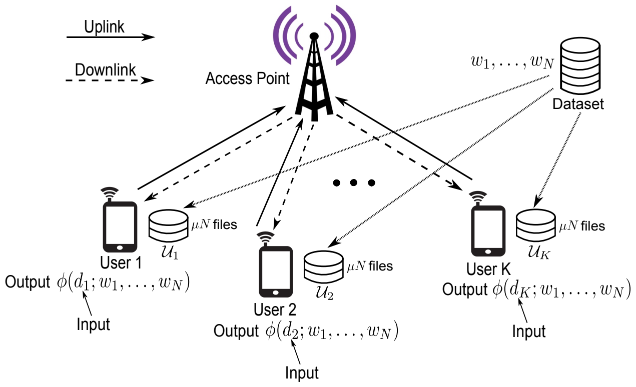


Figure 2.10: A wireless distributed computing system.

The system has a dataset (e.g., a feature repository of objects in a image recognition application) that is evenly partitioned into N files $w_1, \ldots, w_N \in \mathbb{F}_{2^F}$, for some $N, F \in \mathbb{N}$. Each user k has a length-D input $d_k \in \mathbb{F}_{2^D}$ (e.g., user's image in the image recognition application) to process using the N files. To do that, as shown in Figure 2.10, User k needs to compute

$$\phi(\underbrace{d_k}_{\text{input}}; \underbrace{w_1, \ldots, w_N}_{\text{dataset}}), \tag{2.60}$$

where $\phi : \mathbb{F}_{2^D} \times (\mathbb{F}_{2^F})^N \to \mathbb{F}_{2^B}$ is an output function that maps the input d_k to an output result (e.g., the returned result after processing the image) of length $B \in \mathbb{N}$.

We assume that every mobile user has a local memory that can store up to μ fractions of the dataset (i.e., μN files), for some constant parameter μ. We focus on the case where $\frac{1}{K} \le \mu < 1$, such that each user does not have enough storage for the entire dataset, but the entire dataset can be stored collectively across all the users. We denote the set of indices of the files stored by User k as $\mathcal{U}_k$. The selections of $\mathcal{U}_k$s are design parameters, and we denote the design of $\mathcal{U}_1, \ldots, \mathcal{U}_K$ as *dataset placement*. The dataset placement is performed in prior to the computation (e.g., users download parts of the feature repository when installing the image recognition application).

Remark 2.9. The employed physical-layer network model is rather simple and one can do better using a more detailed model and more advanced techniques. However we note that any wireless medium can be converted to our simple model using (1) TDMA on uplink; and (2) broadcast at the rate of weakest user on downlink. Since our goal is to introduce a "coded" framework for scalable wireless distributed computing, we decide to abstract out the physical layer and focus on the amount of data needed to be communicated. □

Distributed computing model. Motivated by prevalent distributed computing structures like MapReduce Dean and Ghemawat, 2004 and Spark Zaharia *et al.*, 2010, we assume that the computation for input d_k can be decomposed as

$$\phi(d_k; w_1, \ldots, w_N) = h(g_1(d_k; w_1), \ldots, g_N(d_k; w_N)), \tag{2.61}$$

where

- The "Map" functions $g_n(d_k; w_n) : \mathbb{F}_{2^D} \times \mathbb{F}_{2^F} \to \mathbb{F}_{2^T}$, $n \in \{1, \ldots, N\}$, $k \in \{1, \ldots, K\}$, maps the input d_k and the file w_n into an *intermediate value* $v_{k,n} = g_n(d_k; w_n) \in \mathbb{F}_{2^T}$, for some $T \in \mathbb{N}$,
- The "Reduce" function $h : (\mathbb{F}_{2^T})^N \to \mathbb{F}_{2^B}$ maps the intermediate values for input d_k in all files into the output value $\phi(d_k; w_1, \ldots, w_N) = h(v_{k,1}, \ldots, v_{k,N})$, for all $k \in \{1, \ldots, K\}$.

We focus on the applications in which the size of the users' inputs is much smaller than the size of the computed intermediate values, i.e., $D \ll T$. As a result, the overhead of disseminating the inputs is negligible, and we assume that the users' inputs $d_1, \ldots, d_K$ are known at each user before the computation starts.

Remark 2.10. The above assumption holds for various wireless distributed computing applications. For example, in a mobile navigation application, an input is simply the address of the intended destination. The computed intermediate results contain all possible routes between the two end locations, from which the fastest one is computed for the user. Similarly, for a set of "filtering" applications like image recognition (or similarly augmented reality) and recommendation systems, the inputs are light-weight queries (e.g., the feature vector of an image) that are much smaller than the filtered intermediate results containing all attributes of related information. For example, an input can be multiple words describing the type of restaurant a user is interested in, and the intermediate results returned by a recommendation system application can be a list of relevant information that include customers' comments, pictures, and videos of the recommended restaurants. □

Following the decomposition in (2.61), the overall computation proceeds in three phases: *Map*, *Shuffle*, and *Reduce*.

Map phase: User k, $k \in \{1, \ldots, K\}$, computes the Map functions of $d_1, \ldots, d_K$ based on the files in $\mathcal{U}_k$. For each input d_k and each file w_n in $\mathcal{U}_k$, User k computes $g_n(d_k, w_n) = v_{k,n}$.

Shuffle phase: Users exchange the needed intermediate values via the access point they all wirelessly connect to. As a result, the Shuffle phase breaks into two sub-phases: *uplink communication* and *downlink communication.*

On the uplink, user k creates a message W_k as a function of the intermediate values computed locally, i.e.,

$$W_k = \psi_k \left(\{ v_{k,n} : k \in \{1, \ldots, K\}, n \in \mathcal{U}_k \} \right), \tag{2.62}$$

and communicates W_k to the access point.

Definition 2.4 (Uplink Communication Load). We define the *uplink communication load*, denoted by L_u, as the total number of bits in all uplink messages $W_1, \ldots, W_K$, normalized by the number of bits in the N intermediate values required by a user (i.e., NT). $\Diamond$

We assume that the access point does not have access to the dataset. Upon decoding all the uplink messages $W_1, \ldots, W_K$, the access point generates a message X from the decoded uplink messages, i.e.,

$$X = \rho(W_1, \ldots, W_K), \tag{2.63}$$

and then broadcasts X to all users on the downlink.

Definition 2.5 (Downlink Communication Load)**.** We define the *downlink communication load*, denoted by L_d, as the number of bits in the downlink message X, normalized by NT. $\Diamond$

Reduce phase: User k, $k \in \{1, \ldots, K\}$, uses the locally computed results $\{\vec{g}_n : n \in \mathcal{U}_k\}$ and the decoded downlink message X to construct the inputs to the corresponding Reduce function, and calculates the output value $\phi(d_k; w_1, \ldots, w_N) = h(v_{k,1}, \ldots, v_{k,N})$.
Example (uncoded scheme). As a benchmark, we consider an uncoded scheme, where each user receives the needed intermediate values sent uncodedly by some other users and forwarded by the access point, achieving the communication loads

$$L_u^{\text{uncoded}}(\mu) = L_d^{\text{uncoded}}(\mu) = \mu K \cdot (\tfrac{1}{\mu} - 1). \tag{2.64}$$

The above communication loads of the uncoded scheme grow with the number of users K, overwhelming the limited spectral resources. In this section, we argue that by utilizing coding at the users and the access point, we can accommodate any number of users with a *constant* communication load. Particularly, we propose in the next subsection a scalable coded wireless distributed computing (CWDC) scheme that achieves minimum possible uplink and downlink communication load simultaneously, i.e.,

$$L_u^{\text{coded}} = L_u^{\text{optimum}} \approx \tfrac{1}{\mu} - 1, \tag{2.65}$$
$$L_d^{\text{coded}} = L_d^{\text{optimum}} \approx \tfrac{1}{\mu} - 1. \tag{2.66}$$

2.3.2 The proposed CWDC scheme

We present the proposed CWDC scheme for the wireless distributed computing system. We first consider the storage size $\mu \in \{\frac{1}{K}, \frac{2}{K}, \ldots, 1\}$

such that $\mu K \in \mathbb{N}$. We assume that N is sufficiently large such that $N = \binom{K}{\mu K}\eta$ for some $\eta \in \mathbb{N}$. [8]

Dataset placement and Map phase execution. We evenly partition the indices of the N files into $\binom{K}{\mu K}$ disjoint batches, each containing the indices of η files. We denote a batch of file indices as $\mathcal{B}_{\mathcal{T}}$, which is labelled by a unique subset $\mathcal{T} \subset \{1, \ldots, K\}$ of size $|\mathcal{T}| = \mu K$. As such defined, we have

$$\{1, \ldots, N\} = \{i : i \in \mathcal{B}_{\mathcal{T}}, \mathcal{T} \subset \{1, \ldots, K\}, |\mathcal{T}| = \mu K\}. \tag{2.67}$$

User k, $k \in \{1, \ldots, K\}$, stores locally all the files whose indices are in $\mathcal{B}_{\mathcal{T}}$ if $k \in \mathcal{T}$. That is,

$$\mathcal{U}_k = \bigcup_{\mathcal{T}:|\mathcal{T}|=\mu K, k \in \mathcal{T}} \mathcal{B}_{\mathcal{T}}. \tag{2.68}$$

As a result, each of the N files is stored by μK distinct users.

Uplink communication. For any subset $\mathcal{W} \subset \{1, \ldots, K\}$, and any $k \notin \mathcal{W}$, we denote the set of intermediate values needed by User k and known *exclusively* by users in $\mathcal{W}$ as $\mathcal{V}_{\mathcal{W}}^k$. More formally:

$$\mathcal{V}_{\mathcal{W}}^k \triangleq \{v_{k,n} : n \in \bigcap_{i \in \mathcal{W}} \mathcal{U}_i, n \notin \bigcup_{i \notin \mathcal{W}} \mathcal{U}_i\}. \tag{2.69}$$

For all subsets $\mathcal{S} \subseteq \{1, \ldots, K\}$ of size $\mu K + 1$:

1. For each User $k \in \mathcal{S}$, $\mathcal{V}_{\mathcal{S} \backslash \{k\}}^k$ is the set of intermediate values that are requested by User k and are in the files whose indices are in the batch $\mathcal{B}_{\mathcal{S} \backslash \{k\}}$, and they are exclusively known at all users whose indices are in $\mathcal{S} \backslash \{k\}$. We evenly and arbitrarily split $\mathcal{V}_{\mathcal{S} \backslash \{k\}}^k$, into μK disjoint segments $\{\mathcal{V}_{\mathcal{S} \backslash \{k\},i}^k : i \in \mathcal{S} \backslash \{k\}\}$, where $\mathcal{V}_{\mathcal{S} \backslash \{k\},i}^k$ denotes the segment associated with User i in $\mathcal{S} \backslash \{k\}$ for User k. That is, $\mathcal{V}_{\mathcal{S} \backslash \{k\}}^k = \bigcup_{i \in \mathcal{S} \backslash \{k\}} \mathcal{V}_{\mathcal{S} \backslash \{k\},i}^k$.

2. User i, $i \in \mathcal{S}$, sends the bit-wise XOR, denoted by $\oplus$, of all the segments associated with it in $\mathcal{S}$, i.e., User i sends the coded segment $W_i^{\mathcal{S}} \triangleq \bigoplus_{k \in \mathcal{S} \backslash \{i\}} \mathcal{V}_{\mathcal{S} \backslash \{k\},i}^k$.

[8]For small number of files $N < \binom{K}{\mu K}$, we can apply the coded wireless distributed computing scheme to a smaller subset of users, achieving a part of the gain in reducing the communication load.

Since the coded message $W_i^{\mathcal{S}}$ contains $\frac{\eta}{\mu K}T$[9] bits for all $i \in \mathcal{S}$, there are a total of $\frac{(\mu K+1)\eta}{\mu K}T$ bits communicated on the uplink in every subset $\mathcal{S}$ of size $\mu K + 1$. Therefore, the uplink communication load achieved by this coded scheme is

$$L_u^{\text{coded}}(\mu) = \frac{\binom{K}{\mu K+1}(\mu K+1)\cdot\eta\cdot T}{\mu K\cdot NT} = \frac{1}{\mu} - 1,\ \mu \in \{\tfrac{1}{K}, \tfrac{2}{K}, \ldots, 1\}. \tag{2.70}$$

Downlink communication. For all subsets $\mathcal{S} \subseteq \{1, \ldots, K\}$ of size $\mu K + 1$, the access point computes μK random linear combinations of the uplink messages generated based on the subset $\mathcal{S}$:

$$C_j^{\mathcal{S}}(\{W_i^{\mathcal{S}} : i \in \mathcal{S}\}),\ j = 1, \ldots, \mu K, \tag{2.71}$$

and multicasts them to all users in $\mathcal{S}$.

Since each linear combination contains $\frac{\eta}{\mu K}T$ bits, the coded scheme achieves a downlink communication load

$$L_d^{\text{coded}}(\mu) = \frac{\binom{K}{\mu K+1}\eta\cdot T}{NT} = \frac{\mu K}{\mu K+1}\cdot(\tfrac{1}{\mu} - 1),\ \mu \in \{\tfrac{1}{K}, \tfrac{2}{K}, \ldots, 1\}. \tag{2.72}$$

After receiving the random linear combinations $C_1^{\mathcal{S}}, \ldots, C_{\mu K}^{\mathcal{S}}$, User i, $i \in \mathcal{S}$, cancels all segments she knows locally, i.e., $\bigcup\limits_{k\in\mathcal{S}\backslash\{i\}}\{\mathcal{V}^k_{\mathcal{S}\backslash\{k\},j} : j \in \mathcal{S}\backslash\{k\}\}$. Consequently, User i obtains μK random linear combinations of the required μK segments $\{\mathcal{V}^i_{\mathcal{S}\backslash\{i\},j} : j \in \mathcal{S}\backslash\{i\}\}$.

When μK is not an integer, we can first expand $\mu = \alpha\mu_1 + (1-\alpha)\mu_2$ as a convex combination of $\mu_1 \triangleq \lfloor \mu K \rfloor / K$ and $\mu_2 \triangleq \lceil \mu K \rceil / K$. Then we partition the set of the N files into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$ of sizes $|\mathcal{I}_1| = \alpha N$ and $|\mathcal{I}_2| = (1-\alpha)N$. We next apply the above coded scheme respectively to the files in $\mathcal{I}_1$ and $\mathcal{I}_2$, yielding the following communication loads.

$$L_u^{\text{coded}}(\mu) = \alpha(\tfrac{1}{\mu_1} - 1) + (1-\alpha)(\tfrac{1}{\mu_2} - 1), \tag{2.73}$$

$$L_d^{\text{coded}}(\mu) = \alpha\frac{\mu_1 K}{\mu_1 K+1}\cdot(\tfrac{1}{\mu_1} - 1) + (1-\alpha)\frac{\mu_2 K}{\mu_2 K+1}\cdot(\tfrac{1}{\mu_2} - 1). \tag{2.74}$$

Hence, for general storage size μ, CWDC achieves the following communication loads.

$$L_u^{\text{coded}}(\mu) = \text{Conv}(\tfrac{1}{\mu} - 1), \tag{2.75}$$

$$L_d^{\text{coded}}(\mu) = \text{Conv}(\tfrac{\mu K}{\mu K+1}\cdot(\tfrac{1}{\mu} - 1)), \tag{2.76}$$

[9]Here we assume that T is sufficiently large such that $\frac{T}{\mu K} \in \mathbb{N}$.

where $\mathrm{Conv}(f(\mu))$ denotes the lower convex envelop of the points $\{(\mu, f(\mu)) : \mu \in \{\frac{1}{K}, \frac{2}{K}, ..., 1\}\}$ for function $f(\mu)$.

We summarize the performance of the proposed CWDC scheme in the following theorem.

Theorem 2.3. For a wireless distributed computing application with a dataset of N files, and K users that each can store $\mu \in \{\frac{1}{K}, \frac{2}{K}, \ldots, 1\}$ fraction of the files, the proposed CWDC scheme achieves the following uplink and downlink communication loads for sufficiently large N.

$$L_u^{\text{coded}}(\mu) = \tfrac{1}{\mu} - 1, \tag{2.77}$$

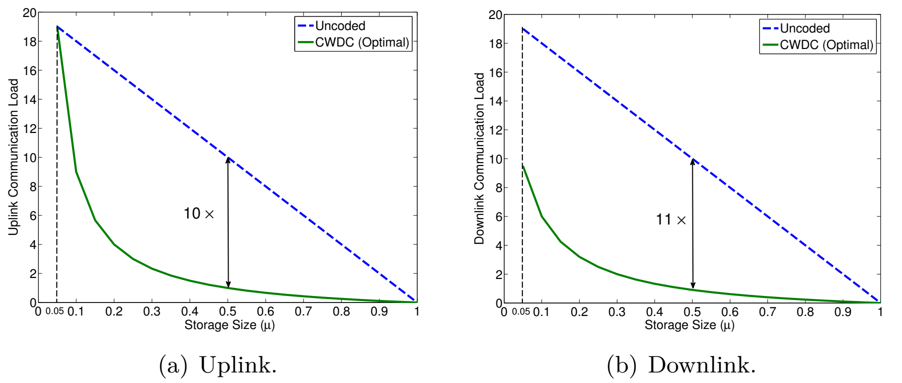
$$L_d^{\text{coded}}(\mu) = \tfrac{\mu K}{\mu K+1} \cdot (\tfrac{1}{\mu} - 1). \tag{2.78}$$

For general $\frac{1}{K} \leq \mu \leq 1$, the achieved loads are as stated in (2.75) and (2.76).

Remark 2.11. Theorem 2.3 implies that, for large K, $L_u^{\text{coded}}(\mu) \approx L_d^{\text{coded}}(\mu) \approx \frac{1}{\mu} - 1$, which is independent of the number of users. Hence, we can accommodate any number of users without incurring extra communication load, and the proposed scheme is *scalable*. The reason for this phenomenon is that, as more users joint the network, with an appropriate dataset placement, we can create coded multicasting opportunities to reduce the communication loads by a factor that scales linearly with K. Such phenomenon was also observed in the context of cache networks (see e.g., Maddah-Ali and Niesen, 2014b). $\square$

Remark 2.12. As illustrated in Figure 2.11, the proposed CWDC scheme utilizes coding at the mobile users and the access point to reduce the uplink and downlink communication load by a factor of μK and $\mu K + 1$ respectively, which scale linearly with the *aggregated* storage size of the system. $\square$

Remark 2.13. Compared with distributed computing over wired servers where we only need to design one data shuffling scheme between servers in Section 2.1, here in the wireless setting we jointly design uplink and downlink shuffling schemes, which minimize both the uplink and downlink communication loads. $\square$



(a) Uplink. (b) Downlink.

Figure 2.11: Comparison of the communication loads achieved by the uncoded scheme with those achieved by the proposed CWDC scheme, for a network of $K = 20$ users. Here the storage size $\mu \geq \frac{1}{K} = 0.05$ such that the entire dataset can be stored across the users.

2.3.3 Optimality of the proposed CWDC scheme

In this subsection, we demonstrate in the following theorem, that the proposed CWDC scheme achieves the minimum uplink and downlink communication loads using any scheme.

Theorem 2.4. For a wireless distributed computing application using any dataset placement and communication schemes that achieve an uplink load L_u and a downlink load L_d, L_u and L_d are lower bounded by $L_u^{\text{coded}}(\mu)$ and $L_d^{\text{coded}}(\mu)$ as stated in Theorem 2.3 respectively.

Remark 2.14. Using Theorem 2.3 and 2.4, we have completely characterized the minimum achievable uplink and downlink communication loads, using *any* dataset placement, uplink and downlink communication schemes. This implies that the proposed CWDC scheme *simultaneously* minimizes both uplink and downlink communication loads required to accomplish distributed computing, and no other scheme can improve upon it. This also demonstrates that there is no fundamental tension between optimizing uplink and downlink communication in wireless distributed computing. □

For a dataset placement $\mathcal{U} = \{\mathcal{U}_k\}_{k=1}^{K}$, we denote the minimum possible uplink and downlink communication loads, achieved by any uplink-downlink communication scheme to accomplish wireless distributed computing, by $L_u^*(\mathcal{U})$ and $L_d^*(\mathcal{U})$ respectively. We next prove Theorem 2.4 by deriving lower bounds on $L_u^*(\mathcal{U})$ and $L_d^*(\mathcal{U})$ respectively.

Lower bound on $L_u^*(\mathcal{U})$

For a given dataset placement $\mathcal{U}$, we denote the number of files that are stored at j users as $a_{\mathcal{U}}^j$, for all $j \in \{1,\dots,K\}$, i.e.,

$$a_{\mathcal{U}}^j = \sum_{\mathcal{J}\subseteq\{1,\dots,K\}:|\mathcal{J}|=j} |(\underset{k\in\mathcal{J}}{\cap}\mathcal{U}_k)\backslash(\underset{i\notin\mathcal{J}}{\cup}\mathcal{U}_i)|. \tag{2.79}$$

For any $\mathcal{U}$, it is clear that $\{a_{\mathcal{U}}^j\}_{j=1}^K$ satisfy

$$\sum_{j=1}^K a_{\mathcal{U}}^j = N, \tag{2.80}$$

$$\sum_{j=1}^K j a_{\mathcal{U}}^j = \mu N K. \tag{2.81}$$

We start the proof with the following lemma, which characterizes a lower bound on $L_u^*(\mathcal{U})$ in terms of the distribution of the files in the dataset placement $\mathcal{U}$, i.e., $a_{\mathcal{U}}^1,\dots,a_{\mathcal{U}}^K$.

Lemma 2.5. $L_u^*(\mathcal{U}) \geq \sum\limits_{j=1}^K \frac{a_{\mathcal{U}}^j}{N}\cdot\frac{K-j}{j}$.

Lemma 2.5 can be proved following the similar steps in the proof of Lemma 2.2 in Section 2.1, after replacing the downlink broadcast message X with the uplink unicast messages $W_1,\dots,W_K$ in conditional entropy terms (since X is a function of $W_1,\dots,W_K$).

Next, since the function $\frac{K-j}{j}$ in Lemma 2.5 is convex in j, and by (2.80) that $\sum\limits_{j=1}^K \frac{a_{\mathcal{U}}^j}{N} = 1$ and (2.81), we have

$$L_u^*(\mathcal{U}) \geq \frac{K-\sum\limits_{j=1}^K j\frac{a_{\mathcal{U}}^j}{N}}{\sum\limits_{j=1}^K j\frac{a_{\mathcal{U}}^j}{N}} = \frac{K-\mu K}{\mu K} = \frac{1}{\mu} - 1. \tag{2.82}$$

We can further improve the lower bound in (2.82) for a particular μ such that $\mu K \notin \mathbb{N}$. For a given storage size μ, we first find two points $(\mu_1, \frac{1}{\mu_1} - 1)$ and $(\mu_2, \frac{1}{\mu_2} - 1)$, where $\mu_1 \triangleq \lfloor \mu K \rfloor / K$ and $\mu_2 \triangleq \lceil \mu K \rceil / K$. Then we find the line $p + qt$ connecting these two points as a function of t, $\frac{1}{K} \leq t \leq 1$, for some constants $p, q \in \mathbb{R}$. We note that p and q are different for different μ and

$$p + qt|_{t=\mu_1} = \frac{1}{\mu_1} - 1, \tag{2.83}$$

$$p + qt|_{t=\mu_2} = \frac{1}{\mu_2} - 1. \tag{2.84}$$

Then by the convexity of the function $\frac{1}{t} - 1$, the function $\frac{1}{t} - 1$ cannot be smaller then the function $p+qt$ at the points $t = \frac{1}{K}, \frac{2}{K}, \ldots, 1$. That is, for all $t \in \{\frac{1}{K}, \ldots, 1\}$,

$$\frac{1}{t} - 1 \geq p + qt. \tag{2.85}$$

By Lemma 2.5, we have

$$\begin{aligned} L_u^*(\mathcal{U}) &\geq \sum_{j=1}^{K} \frac{a_{\mathcal{U}}^j}{N} \cdot \frac{K-j}{j} && (2.86) \\ &= \sum_{t=\frac{1}{K},\ldots,1} \frac{a_{\mathcal{U}}^{tK}}{N} \cdot (\tfrac{1}{t} - 1) && (2.87) \\ &\geq \sum_{t=\frac{1}{K},\ldots,1} \frac{a_{\mathcal{U}}^{tK}}{N} \cdot (p + qt) && (2.88) \\ &= p + q\mu, && (2.89) \end{aligned}$$

Therefore, for general $\frac{1}{K} \leq \mu \leq 1$, $L_u^*(\mathcal{U})$ is lower bounded by the lower convex envelop of the points $\{(\mu, \frac{1}{\mu} - 1) : \mu \in \{\frac{1}{K}, \frac{2}{K}, ..., 1\}\}$.

Lower bound on $L_d^*(\mathcal{U})$

The lower bound on the minimum downlink communication load $L_d^*(\mathcal{U})$ can be proved following the similar steps of lower bounding the minimum uplink communication load $L_u^*(\mathcal{U})$, after making the following enhancements to the downlink communication system:

- We consider the access point as the $(K+1)$th user who has stored all N files and has a virtual input to process. Thus the enhanced downlink communication system has $K+1$ users, and the dataset placement for the enhanced system

$$\bar{\mathcal{U}} \triangleq \{\mathcal{U}, \mathcal{U}_{K+1}\}, \tag{2.90}$$

  where $\mathcal{U}_{K+1}$ is equal to $\{1, \ldots, N\}$.

- We assume that every one of the $K+1$ users can broadcast to the rest of the users, where the broadcast message is generated by mapping the locally stored files.

Apparently the minimum downlink communication load of the system cannot increase after the above enhancements. Thus the lower bound on the minimum downlink communication load of the enhanced system is also a lower bound for the original system.

Then we can apply the same arguments in the proof of Lemma 2.5 to the enhanced downlink system of $K+1$ users, obtaining a lower bound on $L_d^*(\mathcal{U})$, as described in the following corollary:

Corollary 2.6. $L_d^*(\mathcal{U}) \geq \sum_{j=1}^{K} \frac{a_{\mathcal{U}}^j}{N} \cdot \frac{K-j}{j+1}$.

Proof. Applying Lemma 2.5 to the enhanced downlink system yields

$$L_d^*(\bar{\mathcal{U}}) \geq \sum_{j=1}^{K+1} \frac{a_{\bar{\mathcal{U}}}^j}{N} \cdot \frac{K+1-j}{j} \geq \sum_{j=2}^{K+1} \frac{a_{\bar{\mathcal{U}}}^j}{N} \cdot \frac{K+1-j}{j} \tag{2.91}$$

$$= \sum_{j=1}^{K} \frac{a_{\bar{\mathcal{U}}}^{j+1}}{N} \cdot \frac{K-j}{j+1}. \tag{2.92}$$

Since the access point has stored every file, $a_{\bar{\mathcal{U}}}^{j+1} = a_{\mathcal{U}}^j$, for all $j \in \{1, \ldots, K\}$. Therefore, (2.92) can be re-written as

$$L_d^*(\mathcal{U}) \geq L_d^*(\bar{\mathcal{U}}) \geq \sum_{j=1}^{K} \frac{a_{\mathcal{U}}^j}{N} \cdot \frac{K-j}{j+1}. \tag{2.93}$$

∎

Then following the same arguments as in the proof for the minimum uplink communication load, we have

$$L_d^*(\mathcal{U}) \geq \tfrac{K-\mu K}{\mu K+1} = \tfrac{\mu K}{\mu K+1} \cdot (\tfrac{1}{\mu} - 1). \tag{2.94}$$

For general $\frac{1}{K} \leq \mu \leq 1$, $L_d^*(\mathcal{U})$ is lower bounded by the lower convex envelop of the points $\{(\mu, \frac{\mu K}{\mu K+1}(\frac{1}{\mu} - 1)) : \mu \in \{\frac{1}{K}, \frac{2}{K}, ..., 1\}\}$.

This completes the proof of Theorem 2.4.

2.4 Related works and open problems

The problem of characterizing the minimum communication for distributed computing has been previously considered in several settings in both computer science and information theory communities. In Yao, 1979, a basic computing model is proposed, where two parities have x and y and aim to compute a boolean function $f(x, y)$ by exchanging the minimum number of bits between them. Also, the problem of minimizing the required communication for computing the modulo-two sum of distributed binary sources with symmetric joint distribution was introduced in Korner and Marton, 1979. Following these two seminal works, a wide range of communication problems in the scope of distributed computing have been studied (see, e.g., Orlitsky and El Gamal, 1990; Becker and Wille, 1998; Kushilevitz and Nisan, 2006; Orlitsky and Roche, 2001; Nazer and Gastpar, 2007; Ramamoorthy and Langberg, 2013). The key differences distinguishing the setting in this chapter from most of the prior ones are 1) We focus on the flow of communication in a general MapReduce distributed computing framework, rather than the structures of the functions or the input distributions. 2) We do not impose any constraint on the numbers of output results, input data files and computing nodes (they can be arbitrarily large), 3) We do not assume any special property (e.g., linearity) of the computed functions.

The idea of efficiently creating and exploiting *coded multicasting* was initially proposed in the context of cache networks in Maddah-Ali and Niesen, 2014b; Maddah-Ali and Niesen, 2014a, and extended in Ji *et al.*, 2016; Karamchandani *et al.*, 2014, where caches pre-fetch part of the content in a way to enable coding during the content delivery, minimizing the network traffic. In this monograph, we focus on the tradeoff between computation and communication in distributed computing. We demonstrate that the coded multicasting opportunities exploited in the caching problems also exist in the data shuffling of distributed computing frameworks, which can be created by a strategy of repeating the computations of the Map functions specified by the proposed scheme.

There are many follow-up works after the formulation and the characterization of the optimal computation-communication tradeoff in Li *et al.*, 2015; Li *et al.*, 2018b. In Li *et al.*, 2016b, the proposed Coded Distributed Computing (CDC) was utilized in multi-stage computations that consist of executing a series of MapReduce jobs described by a directed acyclic graph, individually minimizing the communication load within each stage. When the Reduce functions are linear, it was shown in Li *et al.*, 2018a that the pre-combining technique (see, e.g., Dean and Ghemawat, 2004) can be combined with CDC to further reduce the bandwidth requirement. When the distributed computing nodes have heterogeneous storage/processing capabilities, or the flexibility of asymmetric computations for the output functions, the optimal task allocation and coded communication schemes were studied in Yu *et al.*, 2017a; Reisizadeh *et al.*, 2017; Kiamari *et al.*, 2017; Ezzeldin *et al.*, 2017. Recent works Konstantinidis and Ramamoorthy, 2018; Woolsey *et al.*, 2018 also studied a new tradeoff between the number of files and the load of communication, under the MapReduce distributed computing framework. Compared with CDC scheme, the new coded computing schemes require exponentially less number of files (or splits of the entire dataset), at the cost of slightly increased communication load.

In another closely related line of works, coded data shuffling schemes were designed to efficiently move data batches between distributed worker, in order to improve the statistical efficiency of distributed iterative algorithms (see, e.g., Lee *et al.*, 2018; Song *et al.*, 2017; Attia and Tandon, 2016; Wan *et al.*, 2018). In this setting, at the beginning of each iteration, the data stored in the local cache memories of the workers are exploited to create coded multicast packets for data shuffling. By the end of the iteration, the workers update their local caches such that efficient multicasting opportunities are enabled in the next iteration.

Finally, we end this chapter with some open problems along the direction of coding for bandwidth reduction.

Heterogeneous networks with asymmetric tasks. It is common to have computing nodes with heterogeneous storage, processing and communication capacities within computer clusters. In addition, processing different parts of the dataset can generate intermediate results with different sizes (e.g., performing data analytics on highly-clustered graphs). For computing over heterogeneous nodes, one solution is to break the more powerful nodes into multiple smaller virtual nodes that have homogeneous capability, and then apply the proposed CDC scheme for the homogeneous setting. When intermediate results have different sizes, the proposed coding scheme still applies, but the coding operations are not symmetric as in the homogeneous case. Alternatively, we can employ a low-complexity greedy approach, in which we assign the Map tasks to maximize the number of multicasting opportunities that simultaneously deliver useful information to the largest possible number of nodes. As mentioned before, some preliminary studies along this direction have been performed to obtain the solutions for some special cases (see, e.g., Yu *et al.*, 2017a; Reisizadeh *et al.*, 2017; Kiamari *et al.*, 2017; Ezzeldin *et al.*, 2017). Nevertheless, systematically characterizing the optimal resource allocation strategies and coding schemes for general heterogeneous networks with asymmetric tasks remains an interesting open problem.

Multi-stage computation tasks. Unlike simple computation tasks like Grep, Join and Sort, many distributed computing applications contain multiples stages of MapReduce computations, whose computation logic can be expressed as a directed acyclic graph. In order to speed up multi-stage computation tasks using codes, while one straightforward approach is to apply the proposed CDC scheme for the cascaded distributed computing framework to compute each stage locally, we expect to achieve a higher reduction in bandwidth consumption and response time by globally designing codes for the entire task graph and accounting for interactions between consecutive stages. A preliminary exploration along this direction was recently presented in Li *et al.*, 2016b.

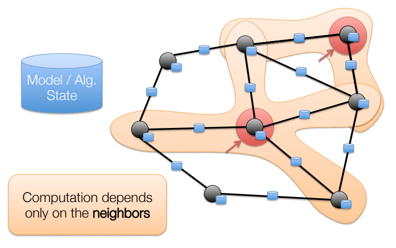
Optimal code design for underlying algebra. In many machine learning applications, the Reduce function has specific algebraic properties that can be exploited in coded computing to substantially reduce the communication load. For example, many Reduce functions in the existing MapReduce frameworks are linear, as they essentially provide an average or a linear summary of the corresponding intermediate computations. As a preliminary work, a coded distributed computing scheme, named "compressed coded distributed computing" (compressed CDC), was proposed in Li *et al.*, 2018a, which incorporates the pre-combining compression techniques into the CDC scheme to further reduce the communication load for linear Reduce functions. Along this direction, it is of great interest to study the optimal task assignment and communication schemes for general non-linear Reduce functions, such as thresholds (max, min, *etc*) and polynomials that are for example used in distributed training.

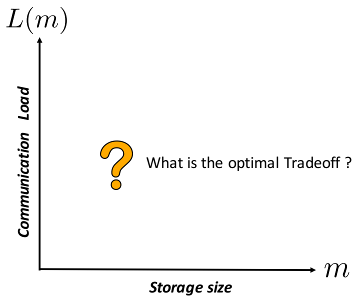
Large-scale graph analytics. There is an increasing interest in executing complex analyses over very large graphs. Examples include graph-theoretic problems in social networks (e.g., targeted advertising and studying the spread of information), bioinformatics (e.g., study of the interactions between various components in a biological system), and networks (e.g., network analysis for intelligence and surveillance. Popular distributed graph computing frameworks, such as Pregel Malewicz *et al.*, 2010 and GraphLab Low *et al.*, 2012, can be viewed as decomposing the intermediate computations in such a way that they only depend on the neighbors at each node (see Figure 2.12(a)). This is commonly referred to as "think like a vertex" computation model, which makes the parallel computations very efficient by leveraging graph topology to reduce the communication load at each iteration of the algorithm.

More formally, we can consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where for each graph node $v \in \{1, 2, \ldots, |\mathcal{V}|\}$, a file w_v is associated with the node. Let $\mathcal{N}^*(v)$ denote the neighbourhood of node v (including v). We can then model the computation at each vertex v as

$$\phi_v(\mathcal{W}_{\mathcal{N}^*(v)}) = h_v(\{g_{v,j}(w_j) : w_j \in \mathcal{W}_{\mathcal{N}^*(v)}\}), \tag{2.95}$$

where the Map function $g_{v,j}(w_j)$ maps the input file w_j into an intermediate value, and the Reduce function $h_v(.)$ maps the intermediate values





(a) (b)

Figure 2.12: a) An overview of "think like a vertex" approach taken in common parallel graph computing frameworks, in which the intermediate computations only depend on the neighbors at each node *Distributed Algorithms and Optimization Lecture Notes* n.d. ; b) Illustration of the fundamental trade-off curve between communication load L and storage size at each server m in parallel graph processing.

of the neighboring nodes of v into the final output value ϕ_v. We note that a key difference between eq. (2.95) and the general MapReduce computation in eq. (2.1) is that the computations at each node now only depend on the neighboring nodes according to the graph topology.

Based on the above abstraction of the computation model, an interesting problem is to design the optimal allocation of the subset of nodes (or data) to each available server and the coding for data shuffling, such that the amount of communication between servers is minimized. More specifically, let us denote the number of available servers by K and assume that the maximum number of nodes that can be assigned to one server or the storage size is denoted by m. Our goal is to characterize the fundamental trade-off curve between communication load and storage size (L, m) for an arbitrary graph, and how coding can help in achieving this fundamental limit (see Figure 2.12(b)). A preliminary exploration for random graphs was recently presented in Prakash *et al.*, 2018.

3

Coding for Straggler Mitigation

Straggling machines cause a major performance bottleneck as distributed computing applications continue to scale out (see, e.g., Zaharia *et al.*, 2008; Ananthanarayanan *et al.*, 2013; Dean and Barroso, 2013). It was recently proposed to use techniques from coding theory to alleviate the effect of stragglers in large-scale data analytics, especially performed over low-end machines on shared platforms like Amazon EC2. The key idea is to inject and leverage redundant computation tasks into the cluster, such that the overall computation task can be accomplished without waiting for the results from the unknown stragglers, hence significantly reducing the overall computation latency.

As a motivating example of this concept, we consider a distributed matrix-vector multiplication problem, which underlies many distributed machine learning algorithms. Given a data matrix $\mathbf{A}$ and a target vector $\mathbf{x}$, the goal is to compute the product $\mathbf{Ax}$ distributedly over 3 workers. The conventional way of doing this is to first partition the matrix $\mathbf{A}$ into 3 sub-matrices $\mathbf{A}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$, such that $\mathbf{A} = [\mathbf{A}_1; \mathbf{A}_2; \mathbf{A}_3]$, and then compute $\mathbf{A}_i\mathbf{x}$ at worker i. Using this approach, the overall computation time is limited by the slowest worker, and can be indefinitely long if one worker becomes irresponsive. Utilizing the idea of erasure coding,

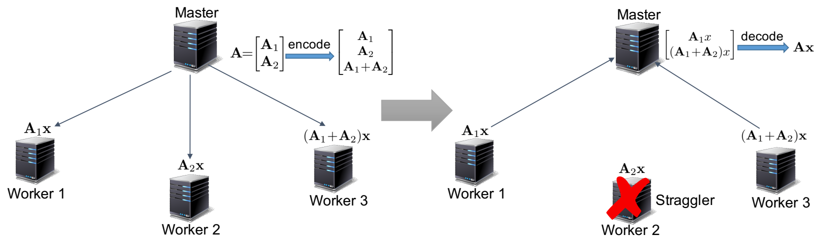

Master
$\mathbf{A}=\begin{bmatrix}\mathbf{A}_1\\ \mathbf{A}_2\end{bmatrix} \xrightarrow{\text{encode}} \begin{bmatrix}\mathbf{A}_1\\ \mathbf{A}_2\\ \mathbf{A}_1+\mathbf{A}_2\end{bmatrix}$
$\mathbf{A}_1\mathbf{x}$
Worker 1
$\mathbf{A}_2\mathbf{x}$
Worker 2
$(\mathbf{A}_1+\mathbf{A}_2)\mathbf{x}$
Worker 3
Master
$\begin{bmatrix}\mathbf{A}_1x\\ (\mathbf{A}_1+\mathbf{A}_2)x\end{bmatrix} \xrightarrow{\text{decode}} \mathbf{A}\mathbf{x}$
$\mathbf{A}_1\mathbf{x}$
Worker 1
$\mathbf{A}_2\mathbf{x}$
Straggler
Worker 2
$(\mathbf{A}_1+\mathbf{A}_2)\mathbf{x}$
Worker 3

Figure 3.1: Coded matrix-vector multiplication. Each worker stores a coded sub-matrix of the data matrix $\mathbf{A}$. During computation, the master can recover the final result using the results of any 2 out of the 3 workers.

as shown in Figure 3.1, a master node partitions the matrix into two sub-matrices $\mathbf{A}_1$ and $\mathbf{A}_2$, and creates a coded sub-matrix $\mathbf{A}_1 + \mathbf{A}_2$, and gives each of these three sub-matrices to one of the workers for computation. Now, the master can recover the desired computation from the results of *any* 2 out of the 3 workers. For example as shown in Figure 3.1, the missing result $\mathbf{A}_2\mathbf{x}$ can be recovered by subtracting the result of worker 1 from that of worker 3. This example illustrates that by introducing 50% redundant computations, we can now tolerate a single straggler. For a general matrix-vector multiplication problem distributedly executed over n workers, it was proposed in Lee *et al.*, 2018 to first partition the matrix into k sub-matrices, for some $k < n$, and then use an (n, k) MDS code (e.g., Reed-Solomon code) to generate n coded sub-matrices, each of which is stored on a worker. During the computation process, each worker multiplies its local sub-matrix with the target vector and returns the result to the master. Due to the "k out of n" property of the (n, k) MDS code, the master can recover the overall computation result using the results from the fastest k worker, protecting the system from as many as $n - k$ stragglers.

While repeating computation tasks has been demonstrated to be an effective approach in straggler mitigation (see, e.g., Ananthanarayanan *et al.*, 2013; Wang *et al.*, 2014; Gardner *et al.*, 2015; Joshi *et al.*, 2017), many recent works have been focusing on characterizing the optimal codes to combat the straggler's effect of distributed linear algebraic computations like matrix-vector and matrix-matrix multiplication. For a problem of multiplying a matrix with a long vector, a sparse code was designed in Dutta *et al.*, 2016 such that only a subset of the entries of the vector are needed for local computations, while still maintaining the robustness to a certain number of stragglers. In the problem of distributed matrix-matrix multiplication where we want to compute the multiplication of two large matrices $\mathbf{A}$ and $\mathbf{B}$ over distributed workers, each of whom can only store a part of $\mathbf{A}$ and $\mathbf{B}$, product code was proposed in Lee *et al.*, 2017 to separately encode $\mathbf{A}$ and $\mathbf{B}$ using MDS codes, and then each worker is assigned to compute the product of a coded sub-matrix of $\mathbf{A}$ and a coded sub-matrix of $\mathbf{B}$. Using the product code, the master needs to wait for a much less number of workers before it can recover the final multiplication results, compared with schemes that only code one of the two matrices. Finally, in Reisizadeh *et al.*, 2017; Aktas *et al.*, 2018, the optimal task/resource allocation (i.e., where and when to launch the redundant coded tasks) was studied to minimize the overall computation latency.

In this chapter, we first consider a distributed matrix-matrix multiplication problem, and propose an optimal coded computation scheme, named "polynomial code" to achieve the minimum possible recovery threshold, which is defined as the number of workers the master needs to wait for before recovering the overall computation result. Next, we go beyond matrix algebra, and consider distributed computing of an arbitrary multivariate polynomial over a dataset. For this problem, we propose "Lagrange Coded Computing" (LCC), which leverages the well-known Lagrange interpolation polynomial to create computation redundancy in a novel coded form across the workers, and achieves the minimum recovery threshold during job execution. We apply LCC to a fundamental machine learning task – least-squares regression, where the gradient computed in each iteration of the gradient descent algorithm is a quadratic function of the training data. For the task of training a regression model on big data, we empirically demonstrate a $2.36\times \sim 12.65\times$ latency reduction over state-of-the-art straggler mitigation techniques. Finally, we end this chapter by discussing coded computation schemes for more general computation tasks (e.g., training neural networks), and some open problems along this research direction.

3.1 Optimal coding for matrix multiplications

In this section, we develop "polynomial code" for distributedly computing large-scale matrix-matrix multiplication. We show that the proposed polynomial code requires the minimum number of workers returning their computation results, and this number does not scale with the network size.

3.1.1 System model, problem formulation, and main result

We consider a problem of matrix multiplication with two input matrices $A \in \mathbb{F}_q^{s\times r}$ and $B \in \mathbb{F}_q^{s\times t}$, for some integers r, s, t and a sufficiently large finite field $\mathbb{F}_q$. We are interested in computing the product $C \triangleq A^{\intercal}B$ in a distributed computing environment with a master node and N worker nodes, where each worker can store $\frac{1}{m}$ fraction of A and $\frac{1}{n}$ fraction of B, for some parameters $m, n \in \mathbb{N}^{+}$ (see Figure 3.2). We assume at least one of the two input matrices A and B is tall (i.e. $s \geq r$ or $s \geq t$), because otherwise the output matrix C would be rank inefficient and the problem is degenerated.

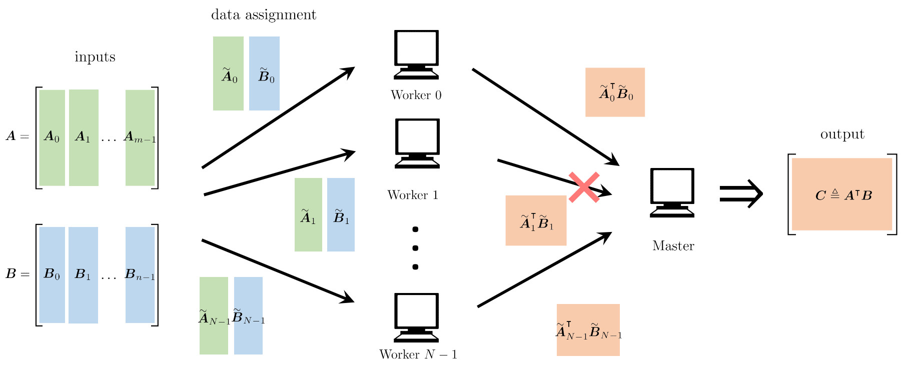

data assignment
inputs
$A =$ A_0 A_1 … A_{m-1}
$B =$ B_0 B_1 … B_{n-1}
$\tilde{A}_0$ $\tilde{B}_0$
$\tilde{A}_1$ $\tilde{B}_1$
$\tilde{A}_{N-1}$ $\tilde{B}_{N-1}$
Worker 0
Worker 1
Worker $N-1$
$\tilde{A}_0^{\intercal}\tilde{B}_0$
$\tilde{A}_1^{\intercal}\tilde{B}_1$
$\tilde{A}_{N-1}^{\intercal}\tilde{B}_{N-1}$
Master
output
$C \triangleq A^{\intercal}B$

Figure 3.2: Overview of the distributed matrix multiplication framework. Coded data are initially stored distributedly at N workers according to data assignment. Each worker computes the product of the two stored matrices and returns it to the master. By carefully designing the computation strategy, the master can decode given the computing results from a subset of workers, without having to wait for the stragglers (worker 1 in this example).

Specifically, each worker i can store two matrices $\tilde{A}_i \in \mathbb{F}_q^{s\times\frac{r}{m}}$ and $\tilde{B}_i \in \mathbb{F}_q^{s\times\frac{t}{n}}$, computed based on *arbitrary functions* of A and B respectively. Each worker can compute the product $\tilde{C}_i \triangleq \tilde{A}_i^\intercal \tilde{B}_i$, and return it to the master. The master waits only for the results from a subset of workers, before proceeding to recover (or compute) the final output C given these products using certain *decoding functions*.[1]

Problem formulation

Given the above system model, we formulate the *distributed matrix multiplication problem* based on the following terminology: We define the *computation strategy* as the $2N$ functions, denoted by

$$\boldsymbol{f} = (f_0, f_1, ..., f_{N-1}), \qquad \boldsymbol{g} = (g_0, g_1, ..., g_{N-1}), \tag{3.1}$$

that are used to compute each $\tilde{A}_i$ and $\tilde{B}_i$. Specifically,

$$\tilde{A}_i = f_i(A), \qquad \tilde{B}_i = g_i(B), \qquad \forall\, i \in \{0, 1, ..., N-1\}. \tag{3.2}$$

For any integer k, we say a computation strategy is *k-recoverable* if the master can recover C given the computing results from *any* k workers. We define the *recovery threshold* of a computation strategy, denoted by $k(\boldsymbol{f}, \boldsymbol{g})$, as the minimum integer k such that computation strategy $(\boldsymbol{f}, \boldsymbol{g})$ is k-recoverable.

Using the above terminology, we define the following concept:

Definition 3.1. For a distributed matrix multiplication problem of computing $A^\intercal B$ using N workers that can each store $\frac{1}{m}$ fraction of A and $\frac{1}{n}$ fraction of B, we define the *optimum recovery threshold*, denoted by K^*, as the minimum achievable recovery threshold among all computation strategies, i.e.

$$K^* \triangleq \min_{\boldsymbol{f},\boldsymbol{g}} k(\boldsymbol{f}, \boldsymbol{g}). \tag{3.3}$$

State-of-the-art schemes. There have been two computing schemes proposed earlier for this problem that leverage ideas from coding theory.

[1] Note that we consider the most general model and do not impose any constraints on the decoding functions. However, any good decoding function should have relatively low computation complexity.

The first one, referred to as *one dimensional MDS code* (*1D MDS code*), was introduced in Lee *et al.*, 2018 and extended in Lee *et al.*, 2017. The 1D MDS code, as illustrated before in Figure 3.1, injects redundancy in only one of the input matrices using maximum distance separable (MDS) codes Singleton, 1964. In general, one can show that the 1D MDS code achieves a recovery threshold of

$$K_{\text{1D-MDS}} \triangleq N - \frac{N}{n} + m = \Theta(N). \tag{3.4}$$

An alternative computing scheme was recently proposed in Lee *et al.*, 2017 for the case of $m = n$, referred to as the *product code*, which instead injects redundancy in both input matrices. This coding technique has also been proposed earlier in the context of Fault Tolerant Computing in Huang and Abraham, 1984; Jou and Abraham, 1986. As shown in Figure 3.3, product code aligns workers in an $\sqrt{N}$–by–$\sqrt{N}$ layout. The matrix A is divided along the columns into m submatrices, encoded using an $(\sqrt{N}, m)$ MDS code into $\sqrt{N}$ coded matrices, and then assigned to the $\sqrt{N}$ columns of workers. Similarly $\sqrt{N}$ coded matrices of B are created and assigned to the $\sqrt{N}$ rows. Given the property of MDS codes, the master can decode an entire row after obtaining any m results in that row; likewise for the columns. Consequently, the master can recover the final output using a peeling algorithm, iteratively decoding the MDS codes on rows and columns until the output C is completely available. For example, if the 5 computing results $A_1^\intercal B_0$, $A_1^\intercal B_1$, $(A_0 + A_1)^\intercal B_1$, $A_0^\intercal(B_0 + B_1)$, and $A_1^\intercal(B_0 + B_1)$ are received as demonstrated in Figure 3.3, the master can recover the needed results by computing $A_0^\intercal B_1 = (A_0 + A_1)^\intercal B_1 - A_1^\intercal B_1$ then $A_0^\intercal B_0 = A_0^\intercal(B_0 + B_1) - A_0^\intercal B_1$. In general, one can show that the product code achieves a recovery threshold of

$$K_{\text{product}} \triangleq 2(m-1)\sqrt{N} - (m-1)^2 + 1 = \Theta(\sqrt{N}), \tag{3.5}$$

which significantly improves over $K_{\text{1D-MDS}}$.

Main Result

Our main result, which demonstrates that the optimum recovery threshold can be far less than what the above two schemes achieve, is stated

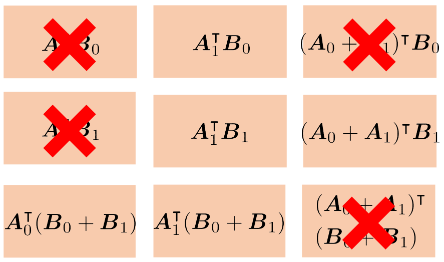

$A_1^\intercal B_0$
$A_1^\intercal B_1$
$(A_0 + A_1)^\intercal B_1$
$A_0^\intercal (B_0 + B_1)$
$A_1^\intercal (B_0 + B_1)$

Figure 3.3: Product code Lee *et al.*, 2017 in an example with $N = 9$ workers that can each store half of A and half of B.

in the following theorem:

Theorem 3.1. For a distributed matrix multiplication problem of computing $A^\intercal B$ using N workers that can each store $\frac{1}{m}$ fraction of A and $\frac{1}{n}$ fraction of B, the minimum recovery threshold K^* is

$$K^* = mn. \tag{3.6}$$

Furthermore, there is a computation strategy, referred to as the *polynomial code*, that achieves the above K^* while allowing efficient decoding at the master node, i.e., with complexity equal to that of polynomial interpolation given mn points.

We prove Theorem 3.1 in the next subsection, where we first describe the proposed polynomial code that achieves a recovery threshold $K_{\text{poly}} = mn$, and then develop an information theoretic converse demonstrating that K^* is lower bounded by mn.

Remark 3.1. Compared to the state of the art Lee *et al.*, 2018; Lee *et al.*, 2017, the polynomial code provides order-wise improvement in terms of the recovery threshold. Specifically, the recovery thresholds achieved by 1D MDS code Lee *et al.*, 2018; Lee *et al.*, 2017 and product code Lee *et al.*, 2017 scale linearly with N and $\sqrt{N}$ respectively, while the proposed polynomial code actually achieves a recovery threshold that does not scale with N. Furthermore, polynomial code achieves the optimal recovery threshold.

Remark 3.2. The polynomial code not only improves the state of the art asymptotically, but also gives strict and significant improvement for any parameter values of N, m, and n (See Figure 3.4 for example).

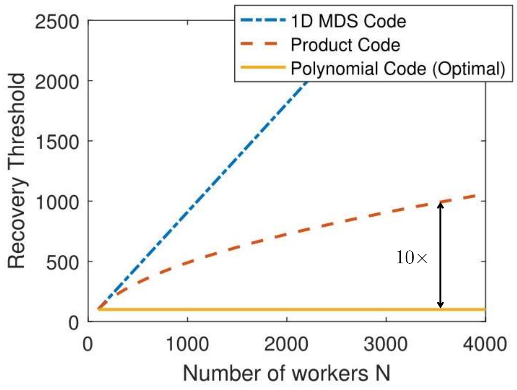


Figure 3.4: Comparison of the recovery thresholds achieved by the proposed polynomial code and the state of the arts (1D MDS code Lee *et al.*, 2018 and product code Lee *et al.*, 2017), where each worker can store $\frac{1}{10}$ fraction of each input matrix. The polynomial code attains the optimum recovery threshold K^*, and significantly improves the state of the art.

Remark 3.3. As we will discuss in Section 3.1.2, decoding polynomial code can be mapped to a polynomial interpolation problem, which can be solved in time almost linear to the input size Kedlaya and Umans, 2011. This is enabled by carefully designing the computing strategies at the workers, such that the computed products form a Reed-Solomon code Roth, 2006 , which can be decoded efficiently using any polynomial interpolation algorithm or Reed-Solomon decoding algorithm that provides the best performance depending on the problem scenario (e.g., Baktir and Sunar, 2006).

3.1.2 Polynomial code and its optimality

In this subsection, we formally describe the polynomial code and its decoding process. We then prove its optimality with an information theoretic converse, which completes the proof of Theorem 3.1. Finally, we demonstrate the optimality of polynomial code under other performance metrics.

Motivating example

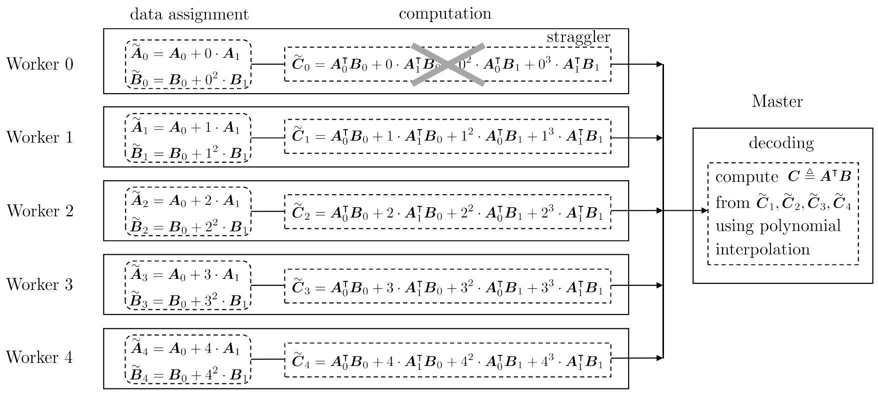

data assignment
computation
straggler
Worker 0
Worker 1
Worker 2
Worker 3
Worker 4
$\tilde{A}_0 = A_0 + 0 \cdot A_1$
$\tilde{B}_0 = B_0 + 0^2 \cdot B_1$
$\tilde{C}_0 = A_0^\intercal B_0 + 0 \cdot A_1^\intercal B_0 + 0^2 \cdot A_0^\intercal B_1 + 0^3 \cdot A_1^\intercal B_1$
$\tilde{A}_1 = A_0 + 1 \cdot A_1$
$\tilde{B}_1 = B_0 + 1^2 \cdot B_1$
$\tilde{C}_1 = A_0^\intercal B_0 + 1 \cdot A_1^\intercal B_0 + 1^2 \cdot A_0^\intercal B_1 + 1^3 \cdot A_1^\intercal B_1$
$\tilde{A}_2 = A_0 + 2 \cdot A_1$
$\tilde{B}_2 = B_0 + 2^2 \cdot B_1$
$\tilde{C}_2 = A_0^\intercal B_0 + 2 \cdot A_1^\intercal B_0 + 2^2 \cdot A_0^\intercal B_1 + 2^3 \cdot A_1^\intercal B_1$
$\tilde{A}_3 = A_0 + 3 \cdot A_1$
$\tilde{B}_3 = B_0 + 3^2 \cdot B_1$
$\tilde{C}_3 = A_0^\intercal B_0 + 3 \cdot A_1^\intercal B_0 + 3^2 \cdot A_0^\intercal B_1 + 3^3 \cdot A_1^\intercal B_1$
$\tilde{A}_4 = A_0 + 4 \cdot A_1$
$\tilde{B}_4 = B_0 + 4^2 \cdot B_1$
$\tilde{C}_4 = A_0^\intercal B_0 + 4 \cdot A_1^\intercal B_0 + 4^2 \cdot A_0^\intercal B_1 + 4^3 \cdot A_1^\intercal B_1$
Master
decoding
compute $C \triangleq A^\intercal B$
from $\tilde{C}_1, \tilde{C}_2, \tilde{C}_3, \tilde{C}_4$
using polynomial
interpolation

Figure 3.5: Example using polynomial code, with $N = 5$ workers that can each store half of each input matrix. (a) Computation strategy: each worker i stores $A_0 + iA_1$ and $B_0 + i^2 B_1$, and computes their product. (b) Decoding: master waits for results from *any* 4 workers, and decodes the output using fast polynomial interpolation algorithm.

We start by demonstrating the key ideas of polynomial code through a motivating example. Consider a distributed matrix multiplication task of computing $C = A^\intercal B$ using $N = 5$ workers that can each store half of the matrices (see Figure 3.5). We evenly divide each input matrix along the column side into 2 submatrices:

$$A = [A_0 \ A_1], \qquad B = [B_0 \ B_1]. \tag{3.7}$$

Given this notation, we essentially want to compute the following 4 uncoded components:

$$C = A^\intercal B = \begin{bmatrix} A_0^\intercal B_0 & A_0^\intercal B_1 \\ A_1^\intercal B_0 & A_1^\intercal B_1 \end{bmatrix}. \tag{3.8}$$

Now we design a computation strategy to achieve the optimum recovery threshold of 4. Suppose elements of A, B are in $\mathbb{F}_7$, let each worker $i \in \{0, 1, ..., 4\}$ store the following two coded submatrices:

$$\tilde{A}_i = A_0 + iA_1, \qquad \tilde{B}_i = B_0 + i^2 B_1. \tag{3.9}$$

To prove that this design gives a recovery threshold of 4, we need to design a valid decoding function for any subset of 4 workers. Without loss of generality, we assume that the master receives the computation results from workers 1, 2, 3, and 4, as shown in Figure 3.5.

According to the designed computation strategy, we have

$$\begin{bmatrix} \tilde{C}_1 \\ \tilde{C}_2 \\ \tilde{C}_3 \\ \tilde{C}_4 \end{bmatrix} = \begin{bmatrix} 1^0 & 1^1 & 1^2 & 1^3 \\ 2^0 & 2^1 & 2^2 & 2^3 \\ 3^0 & 3^1 & 3^2 & 3^3 \\ 4^0 & 4^1 & 4^2 & 4^3 \end{bmatrix} \begin{bmatrix} A_0^\intercal B_0 \\ A_1^\intercal B_0 \\ A_0^\intercal B_1 \\ A_1^\intercal B_1 \end{bmatrix}. \tag{3.10}$$

The coefficient matrix in the above equation is Vandermonde, and hence invertible since its parameters $1, 2, 3, 4$ are distinct in $\mathbb{F}_7$. So one way to recover C is to directly invert equation (3.10). However, directly computing this inverse using the classical inversion algorithm might be expensive in more general cases. Quite interestingly, the decoding process can also be viewed as a polynomial interpolation problem (or equivalently, decoding a Reed-Solomon code subject to erasures).

Specifically, in this example each worker i returns

$$\tilde{C}_i = \tilde{A}_i^\intercal \tilde{B}_i = A_0^\intercal B_0 + iA_1^\intercal B_0 + i^2 A_0^\intercal B_1 + i^3 A_1^\intercal B_1, \tag{3.11}$$

which is essentially the value of the following polynomial at point $x = i$:

$$h(x) \triangleq A_0^\intercal B_0 + xA_1^\intercal B_0 + x^2 A_0^\intercal B_1 + x^3 A_1^\intercal B_1. \tag{3.12}$$

Hence, recovering C using computation results from 4 workers is equivalent to interpolating a degree-3 polynomial given its values at 4 points, and we will later show that this can be performed with almost-linear complexity.

General polynomial code

Now we proceed to present the polynomial code in a general setting that achieves the optimum recovery threshold stated in Theorem 3.1 for any parameter values of N, m, and n. First of all, we evenly divide each input matrix along the column side into m and n submatrices respectively, i.e.,

$$A = [A_0 \ A_1 \ \ldots \ A_{m-1}], \qquad B = [B_0 \ B_1 \ \ldots \ B_{n-1}], \tag{3.13}$$

We then assign each worker $i \in \{0, 1, ..., N-1\}$ a distinct number in $\mathbb{F}_q$, denoted by x_i. Under this setting, we define the following class of computation strategies.

Definition 3.2. Given parameters $\alpha, \beta \in \mathbb{N}$, we define the (α, β)-polynomial code as

$$\tilde{A}_i = \sum_{j=0}^{m-1} A_j x_i^{j\alpha}, \qquad \tilde{B}_i = \sum_{j=0}^{n-1} B_j x_i^{j\beta}, \qquad \forall\ i \in \{0, 1, ..., N-1\}. \tag{3.14}$$

In an (α, β)-polynomial code, each worker i essentially computes

$$\tilde{C}_i = \tilde{A}_i^\intercal \tilde{B}_i = \sum_{j=0}^{m-1}\sum_{k=0}^{n-1} A_j^\intercal B_k x_i^{j\alpha+k\beta}. \tag{3.15}$$

In order for the master to recover the output given any mn results (i.e. achieve the optimum recovery threshold), we carefully select the design parameters α and β, while making sure that no two terms in the above formula has the same exponent of x. One such choice is $(\alpha, \beta) = (1, m)$, i.e,

$$\tilde{A}_i = \sum_{j=0}^{m-1} A_j x_i^{j}, \qquad \tilde{B}_i = \sum_{j=0}^{n-1} B_j x_i^{jm}. \tag{3.16}$$

Hence, each worker essentially computes the value of the following degree $mn-1$ polynomial at point $x = x_i$:

$$h(x) \triangleq \sum_{j=0}^{m-1}\sum_{k=0}^{n-1} A_j^\intercal B_k x^{j+km}, \tag{3.17}$$

where the coefficients are exactly the mn uncoded components of C. Since all x_i's are selected to be distinct, recovering C given results from any mn workers is essentially interpolating $h(x)$ using mn distinct points. Since $h(x)$ has degree $mn-1$, the output C can always be uniquely decoded.

In terms of complexity, this decoding process can be viewed as interpolating degree $mn-1$ polynomials of $\mathbb{F}_q$ for $\frac{rt}{mn}$ times. It is well known that polynomial interpolation of degree k has a complexity of $O(k \log^2 k \log\log k)$ Kedlaya and Umans, 2011. Therefore, decoding polynomial code also only requires a complexity of $O(rt \log^2(mn) \log\log(mn))$. Furthermore, this complexity can be reduced by simply swapping in any faster polynomial interpolation algorithm or Reed-Solomon decoding algorithm.

Remark 3.4. We can naturally extend polynomial code to the scenario where input matrix elements are real or complex numbers. In practical implementation, to avoid handling large elements in the coefficient matrix, we can first quantize input values into numbers of finite digits, embed them into a finite field that covers the range of possible values of the output matrix elements, and then directly apply polynomial code. By embedding into finite fields, we avoid large intermediate computing results, which effectively saves storage and computation time, and reduces numerical errors.

Optimality of polynomial code for recovery threshold

So far we have constructed a computing scheme that achieves a recovery threshold of mn, which upper bounds K^*. To complete the proof of Theorem 3.1, here we establish a matching lower bound through an information theoretic converse.

We need to prove that for any computation strategy, the master needs to wait for at least mn workers in order to recover the output. Recall that at least one of A and B is a tall matrix. Without loss of generality, assume A is tall (i.e. $s \geq r$). Let A be an arbitrary fixed full-rank matrix and B be sampled from $\mathbb{F}_q^{s\times t}$ uniformly at random. It is easy to show that $C = A^\intercal B$ is uniformly distributed on $\mathbb{F}_q^{r\times t}$. This means that the master essentially needs to recover a random variable with entropy of $H(C) = rt \log_2 q$ bits. Note that each worker returns $\frac{rt}{mn}$ elements of $\mathbb{F}_q$, providing at most $\frac{rt}{mn} \log_2 q$ bits of information. Consequently, using a cut-set bound around the master, we can show that at least mn results from the workers need to be collected, and thus we have $K^* \geq mn$.

Remark 3.5 (Random linear code)**.** We conclude this subsection by noting that, another computation design is to let each worker store two random linear combinations of the input submatrices. Although this design can achieve the optimal recovery threshold with high probability, it creates a large coding overhead and requires high decoding complexity (e.g., $O(m^3n^3 + mnrt)$ using the classical inversion decoding algorithm). Compared to random linear code, the proposed polynomial code achieves the optimum recovery threshold deterministically, with a significantly lower decoding complexity.

Optimality of polynomial code for other performance metrics

In the previous subsection, we proved that polynomial code is optimal in terms of the recovery threshold. As a by-product, we can prove that it is also optimal in terms of some other performance metrics. In particular, we consider the following three metrics considered in prior works, and establish the optimality of polynomial code for each of them.
Computation latency is considered in models where the computation time T_i of each worker i is a random variable with a certain probability distribution (e.g, Lee *et al.*, 2018; Lee *et al.*, 2017). The computation latency is defined as the amount of time required for the master to collect enough information to decode C.

Theorem 3.2. For any computation strategy, the computation latency T is always no less than the latency achieved by polynomial code, denoted by T_{poly}. Namely,

$$T \geq T_{\text{poly}}. \tag{3.18}$$

Proof sketch. We know that by the converse proof of Theorem 3.1 that using an arbitrary computation strategy, in order for the master to recover the output matrix C at time T, it has to receive the computation results from at least mn worker. However, using the polynomial code, the matrix C can be recovered as soon as mn workers return their results. Therefore, we have $T \geq T_{\text{poly}}$. □

Probability of failure given a deadline is defined as the probability that the master does not receive enough information to decode C at any time t Dutta *et al.*, 2017.

Corollary 3.3. For any computation strategy, let T denote its computation latency, and let T_{poly} denote the computation latency of polynomial code. We have

$$\mathbb{P}(T > t) \geq \mathbb{P}(T_{\text{poly}} > t) \qquad \forall\, t \geq 0. \tag{3.19}$$

Corollary 3.3 directly follows from Theorem 3.2 since (3.18) implies (3.19) .

Communication load is another important metric in distributed computing (e.g. Li *et al.*, 2015; Li *et al.*, 2018b; Yu *et al.*, 2017a), defined as the minimum number of bits needed to be communicated in order to complete the computation.

Theorem 3.4. Polynomial code achieves the minimum communication load for distributed matrix multiplication, which is given by

$$L^* = rt \log_2 q. \tag{3.20}$$

Proof. Recall that in the converse proof of Theorem 3.1, we have shown that if the input matrices are sampled based on a certain distribution, then decoding the output C requires that the entropy of the entire message received by the server is at least $rt \log_2 q$. Consequently, it takes at least $rt \log_2 q$ bits deliver such messages, which lower bounds the minimum communication load.

On the other hand, the polynomial code requires delivering rt elements in $\mathbb{F}_q$ in total, which achieves this minimum communication load. Hence, the minimum communication load L^* equals $rt \log_2 q$. $\square$

Remark 3.6. While polynomial codes provide the optimal design, with respect to the above metrics, for straggler mitigation in distributed matrix multiplication, one can also consider other metrics and variations of the problem setting for which the problem is still not completely solved. One variation is "approximate distributed matrix multiplication", which has been studied in Gupta *et al.*, 2018; Jahani-Nezhad and Maddah-Ali, 2019. Another variation is coded computing in heterogeneous and dynamic network settings, which has been studied in Reisizadeh *et al.*, 2017; Reisizadeh *et al.*, 2019; Narra *et al.*, 2019; Mallick *et al.*, 2019; Yang *et al.*, 2019; Ferdinand and Draper, 2018.

3.2 Optimal coding for polynomial evaluations

In this section, we go beyond matrix algebra to study the impact of coding on minimizing the recovery threshold in distributed computation of arbitrary multivariate polynomials.

3.2.1 Problem formulation and motivating examples

We consider a problem of evaluating a function f over a dataset $X = (X_1, \ldots, X_K)$. In particular, each X_i is an element in a vector space $\mathbb{V}$ over a field $\mathbb{F}$, and the goal is to compute $Y_1 \triangleq f(X_1), \ldots, Y_K \triangleq f(X_K)$ given the function $f : \mathbb{V} \to \mathbb{U}$, where $\mathbb{U}$ is a vector space over the same field $\mathbb{F}$. The function f can be *any multivariate polynomial* with *vector coefficients*, and we define the degree of the chosen f, denoted by $\deg f$, as the *total degree* of the polynomial.[2]

The computation is carried out in a distributed system consisting of a master and N workers. Each worker has already stored a fraction of the dataset prior to the computation, in a possibly coded manner. Specifically, for each $i \in [N] \triangleq \{1, \ldots, N\}$, worker i stores $\tilde{X}_i \triangleq g_i(X_1, \ldots, X_K)$, where $g_i : \mathbb{V}^K \to \mathbb{V}$ is the encoding function of worker i. We focus on the class of *linear encoding strategies*, meaning that each $\tilde{X}_i$ is a linear combination of $X_1, \ldots, X_K$. This class of encoding designs guarantees low encoding complexity and simple implementation.

During the computation, each worker i computes $\tilde{Y}_i \triangleq f(\tilde{X}_i)$, and returns the result back to the master upon its completion. The master only waits for a fastest subset of workers, until all the final outputs $Y_1, \ldots, Y_K$ can be decoded from the available results by computing their *linear combinations*.[3] Similarly as before, we define the *recovery*

[2]The *total degree* of a polynomial f is the maximum among all the total degrees of its monomials. In the case where $\mathbb{F}$ is finite, we resort to the canonical representation of polynomials, in which the individual degrees within each term is no more than $(|\mathbb{F}| - 1)$.

[3]Note that if the number of workers is too small, obviously no valid computation design exists unless f is a constant. Hence, in the rest of this section we focus on meaningful cases where N is large enough such that there is a valid computation design for at least one non-trivial function f (i.e., $N \geq K$).

threshold as the minimum number of responses that guarantees the completion of the computation task.

This computation model suites the common scenario where a function of interest is of the form $F(X_1, \ldots, X_k) = g(f(X_1), \ldots, f(X_k))$, where f is a "hard to compute" function and g is an "easy to compute" one. This is in accordance with common distributed computing tasks like matrix multiplication, the MapReduce algorithm, and gradient computation.

Based on this setting, the coded computing problem is then formulated as designing the optimal encoding of the dataset (i.e., designing g_i's) over which workers carry out their computations, in order to achieve the minimum recovery threshold, denoted by K^*.

The above computation framework encapsulates many computation tasks of interest, which we highlight in the following examples.

Linear computation. Consider the computation scenario of matrix-vector multiplication $A\vec{b}$, for some dataset $A = \{A_i\}_{i=1}^K$ and some vector $\vec{b}$. This scenario naturally arises in many machine learning algorithms, such as each iteration of linear regression. Our formulation covers this setting by letting $\mathbb{V}$ be the space of matrices of certain dimensions over $\mathbb{F}$, $\mathbb{U}$ be the space of vectors of a certain length over $\mathbb{F}$, X_i be A_i, and $f(X_i) = X_i \cdot \vec{b}$ for all $i \in [K]$. Coded computing for such linear computations has also been studied in Lee *et al.*, 2018; Dutta *et al.*, 2016; Karakus *et al.*, 2017; Wang *et al.*, 2018b.

Bilinear computation. Another computation task of interest, is to evaluate the element-wise products $\{A_i \cdot B_i\}_{i=1}^K$ given two lists of matrices $\{A_i\}_{i=1}^K$ and $\{B_i\}_{i=1}^K$. This computation is the key building block for various algorithms, such as fast matrix multiplication in distributed systems Yu *et al.*, 2017b; Fahim *et al.*, 2017; Yu *et al.*, 2018b. Our formulation covers this setting by letting $\mathbb{V}$ be the space of pairs of two matrices of certain dimensions, $\mathbb{U}$ be the space of matrices of dimension which equals that of the product of the pairs of matrices, $X_i = (A_i, B_i)$, and $f(X_i) = A_i \cdot B_i$ for all $i \in [K]$.

General Tensor algebra. Beyond bilinear operations, distributed computations of multivariate polynomials of larger degree, such as general tensor algebraic functions (i.e. functions composed of inner products, outer products, and tensor contractions) Renteln, 2013, also arise in practice. A specific example is to compute the coordinate transformation of a third-order tensor field at K locations, where given a list of matrices $\{Q^{(i)}\}_{i=1}^{K}$ and a list of third order tensors $\{T^{(i)}\}_{i=1}^{K}$ with matching dimension on each index, the goal is to compute another list of tensors, denoted by $\{T'^{(i)}\}_{i=1}^{K}$, of which each entry is defined as $T'^{(i)}_{j'k'\ell'} \triangleq \sum_{j,k,\ell} T'^{(i)}_{jk\ell} Q^{(i)}_{jj'} Q^{(i)}_{kk'} Q^{(i)}_{\ell\ell'}$. Our formulation covers all functions within this class by letting $\mathbb{V}$ the space of input tensors, $\mathbb{U}$ the space of output tensors, X_i be the inputs, and f be the tensor function.

Gradient computation. Another general class of functions arises from gradient decent algorithms and their variants, which are the workhorse of today's learning tasks. The computation task for this class of functions is to consider one iteration of the gradient decent algorithm, and to evaluate the gradient of the empirical risk $\nabla L_{\mathcal{S}}(h) \triangleq \text{avg}_{z\in\mathcal{S}} \nabla \ell_h(z)$, given a hypothesis $h : \mathbb{R}^d \to \mathbb{R}$, a respective loss function $\ell_h : \mathbb{R}^{d+1} \to \mathbb{R}$, and a training set $\mathcal{S} \subseteq \mathbb{R}^{d+1}$, where d is the number of features. In practice, this computation is carried out by partitioning $\mathcal{S}$ into K subsets $\{\mathcal{S}_i\}_{i=1}^{K}$ of equal sizes, evaluating the partial gradients $\{\nabla L_{\mathcal{S}_i}(h)\}_{i=1}^{K}$ distributedly, and computing the final result using $\nabla L_{\mathcal{S}}(h) = \text{avg}_{i\in[K]} \nabla L_{\mathcal{S}_i}(h)$. We present a specific example of applying this computing model to least-squares regression problems in Section 3.2.5.

3.2.2 Main results and comparison with prior works

We characterize the minimum possible recovery threshold for the above distributed computing problem in the following theorem.

Theorem 3.5. For the above described problem of distributedly evaluating a multivariate polynomial $f : \mathbb{V} \to \mathbb{U}$ on a dataset of K inputs by using N workers, the minimum recovery threshold is given by

$$K^* = (K-1)\deg f + 1$$

when $N \geq K \deg f - 1$, and $K^* = N - \lfloor N/K \rfloor + 1$ otherwise.

We propose a coded computing scheme, named "Lagrange Coded Computing" to achieve this minimum value.

To prove Theorem 3.5, we present the proposed Lagrange Coded Computing (LCC) scheme and characterize its recovery threshold in the next subsection. Moreover, we complete the proof by demonstrating the optimality of Lagrange Coded Computing through a matching converse in Section 3.2.4.

Remark 3.7. LCC generalizes several previously studied scenarios. For example, having $\mathbb{V} = \mathbb{U}$ and f the identity function reduces to the well-studied case of *distributed storage*, in which Theorem 3.5 is well-known (e.g., the Singleton bound Roth, 2006, Theorem 4.1). Further, as previously mentioned, f can correspond to matrix-vector and matrix-matrix multiplication, in which the special cases of Theorem 3.5 are known as well Lee *et al.*, 2018; Yu *et al.*, 2018b. However, LCC substantially generalizes the state of the arts to any computation that can be represented as an arbitrary multivariate polynomial of the input dataset, including many computation scenarios of interest in machine learning.

Remark 3.8. The key idea of LCC is to encode the input dataset using the well-known Lagrange polynomial. In particular, the encoding functions (i.e., g_i's) amount to evaluations of a Lagrange polynomial of degree $K - 1$ at N distinct points. Hence, the computations at the workers amount to evaluations of a *composition* of that polynomial with the desired function f. Therefore, K^* may simply be seen as the number of evaluations that are necessary and sufficient in order to interpolate the composed polynomial, that is later evaluated at certain point to finalize the computation.

Remark 3.9. LCC has a number of additional properties of interest. First, the proposed encoding is *identical* to all multivariate polynomials, which allows pre-encoding of the data without knowing the identity of the computing task. In other words, data encoding of LCC can be *universally* used for any polynomial computation. This is in stark contrast to previous task specific coding techniques in the literature. Furthermore, workers apply the same computation as if no coding took place; a feature that reduces computational costs, and prevents ordinary servers from carrying the burden of outliers. Second, decoding and encoding build upon polynomial interpolation and evaluation, and hence efficient off-the-shelf subroutines can be used.

3.2.3 Lagrange Coded Computing

Illustrative example

Consider a problem of evaluating the quadratic function $f(\boldsymbol{X}_i) = \boldsymbol{X}_i^\top(\boldsymbol{X}_i\boldsymbol{w} - \boldsymbol{y})$, where the input $\boldsymbol{X}_i$'s are real matrices with certain dimensions, and $\boldsymbol{w}$, $\boldsymbol{y}$ are constant vectors with matching lengths. This function naturally appears in gradient computing problems, given that each $f(\boldsymbol{X}_i)$ is the gradient of a commonly used quadratic loss function $(\boldsymbol{X}_i\boldsymbol{w} - \boldsymbol{y})^2$ (with respect to $\boldsymbol{w}$).

We demonstrate Lagrange Coded Computing (LCC) in the scenario where the input data $\boldsymbol{X}$ is partitioned into $K = 2$ batches, and the computing system has $N = 5$ workers. Note that the conventional uncoded repetition design only achieves a recovery threshold of 4. This is since in uncoded repetition one essentially must have $\tilde{\boldsymbol{X}}_1 = \tilde{\boldsymbol{X}}_2 = \boldsymbol{X}_1$ and $\tilde{\boldsymbol{X}}_3 = \tilde{\boldsymbol{X}}_4 = \tilde{\boldsymbol{X}}_5 = \boldsymbol{X}_2$, and clearly, the computation cannot be completed from the results of workers $3, 4$, and 5. However, optimal recovery threshold of 3 is attainable by LCC.

As mentioned earlier, the main idea of LCC is to encode data using a Lagrange polynomial u. To this end, let $u(z) \triangleq \boldsymbol{X}_1 \cdot \frac{z-2}{1-2} + \boldsymbol{X}_2 \cdot \frac{z-1}{2-1} = z(\boldsymbol{X}_2 - \boldsymbol{X}_1) + 2\boldsymbol{X}_1 - \boldsymbol{X}_2$, and observe that $u(1) = \boldsymbol{X}_1$ and $u(2) = \boldsymbol{X}_2$. Then, node i stores $u(i)$, i.e.,

$$\left(\tilde{\boldsymbol{X}}_1, \ldots, \tilde{\boldsymbol{X}}_5\right) = (\boldsymbol{X}_1, \boldsymbol{X}_2) \cdot \begin{pmatrix} 1 & 0 & -1 & -2 & -3 \\ 0 & 1 & 2 & 3 & 4 \end{pmatrix}.$$

Note that when applying f over its stored data, each worker essentially evaluates a linear combination of 6 possible terms: four quadratic $\boldsymbol{X}_i^\top\boldsymbol{X}_j\boldsymbol{w}$ and two linear $\boldsymbol{X}_i^\top\boldsymbol{y}$. However, the master only wants two specific linear combinations of them: $\boldsymbol{X}_1^\top(\boldsymbol{X}_1\boldsymbol{w} - \boldsymbol{y})$ and $\boldsymbol{X}_2^\top(\boldsymbol{X}_2\boldsymbol{w} - \boldsymbol{y})$. Interestingly, LCC optimally aligns the computation of the workers in a sense that the linear combinations returned by the workers belong to a subspace of only 3 dimensions, which can be recovered from the computing results of any 3 workers, while containing the two needed linear combinations.

More specifically, each worker i evaluates the polynomial

$$f(u(z))=(z(\boldsymbol{X}_2-\boldsymbol{X}_1)+2\boldsymbol{X}_1-\boldsymbol{X}_2)^{\top}((z(\boldsymbol{X}_2-\boldsymbol{X}_1)+2\boldsymbol{X}_1-\boldsymbol{X}_2)\boldsymbol{w}-\boldsymbol{y})$$

at $z=i$. Since $f(u(z))$ is a quadratic polynomial, it can be determined given the computation results from *any* three nodes. Furthermore, after decoding the polynomial $f(u(z))$, the master can obtain $f(\boldsymbol{X}_1)$ and $f(\boldsymbol{X}_2)$ by evaluating it at $z=1$ and $z=2$.

General description

When the number of workers is small (i.e, $N < K \deg f - 1$), the optimum recovery threshold $K^* = N - \lfloor N/K \rfloor + 1$ can be easily achieved by uncoded repetition design – that is, by replicating every X_i between $\lfloor N/K \rfloor$ and $\lceil N/K \rceil$ times, it is readily verified that every set of $N - \lfloor N/K \rfloor + 1$ computation results contains at least one copy of $f(X_i)$ for every i. Hence, we focus on the case where $N \geq K \deg f - 1$.

First, we select any K distinct elements $\beta_1, \ldots, \beta_K$ from $\mathbb{F}$, and find a polynomial $u : \mathbb{F} \to \mathbb{V}$ of degree $K-1$ such that $u(\beta_i) = X_i$ for any $i \in [K] = \{1, \ldots, K\}$. This is simply accomplished by letting u be the respective *Lagrange interpolation polynomial* $u(z) \triangleq \sum_{j \in [K]} X_j \cdot \prod_{k \in [K]\setminus\{j\}} \frac{z-\beta_k}{\beta_j-\beta_k}$. We then select N distinct elements $\alpha_1, \ldots, \alpha_N$ from $\mathbb{F}$, and encode the input variables by letting $\tilde{X}_i = u(\alpha_i)$ for any $i \in [N]$. That is,

$$\tilde{X}_i = g_i(X) = u(\alpha_i) \triangleq \sum_{j=1}^{K} X_j \cdot \prod_{k \in [K]\setminus\{j\}} \frac{\alpha_i - \beta_k}{\beta_j - \beta_k}. \tag{3.21}$$

When each worker i computes $\tilde{Y}_i = f(\tilde{X}_i)$, it is essentially evaluating the composition of the two polynomials f and u at point α_i (i.e., $f(u(\alpha_i))$). Note that the composition $f(u(z))$ is also a polynomial, whose degree is $(K-1)\deg f$. Hence, any $(K-1)\deg f + 1$ workers return the evaluations of this polynomial at $(K-1)\deg f + 1$ points, and thus it is recoverable.

Finally, the master aims to recover $f(u(\beta_i)) = f(X_i)$ for all $i \in [K]$, which is possible given that $f(u(z))$ is determined.

Remark 3.10. Note that by choosing $\{\beta_i\}_{i=1}^K = \{\alpha_i\}_{i=1}^K$, the first K workers exactly compute the K required results respectively. This provides a systematic coding design in the sense that the first K workers are the *systematic nodes* and the rest of the $N-K$ workers are *parity nodes*.

Remark 3.11. In our construction, the only restriction imposed on the underlying field is that we need to be able to select N distinct elements $\{\alpha_i\}_{i\in[N]}$. Hence, LCC can be applied over any infinite field or any finite field with at least N elements.

Remark 3.12. In terms of encoding and decoding complexities, the decoding process of LCC is essentially computing the Lagrange interpolation for the polynomial $f \circ u$ at K points. This interpolation can be efficiently computed with an almost linear complexity (i.e., $O(K^* \log^2 K^* \log\log K^*)$ linear operations in $\mathbb{U}$), using fast polynomial arithmetic algorithms Kedlaya and Umans, 2011. Similar to the polynomial code, this decoding complexity can be reduced by simply swapping in any faster interpolation algorithm or Reed-Solomon decoding algorithm.

3.2.4 Optimality of Lagrange Coded Computing

While the analysis of the proposed Lagrange Coded Computing scheme provides an upper bound on the minimum recovery threshold K^*, we now complete the proof of Theorem 3.5 by establishing a matching lower bound of K^* for any polynomial function $f : \mathbb{V} \to \mathbb{U}$.

The proof consists of two steps. In Step 1, we prove the converse for the special case where f is a multilinear function (i.e., f is linear in each variable, where the rest of the variables are fixed). Then in Step 2, we generalize this result to arbitrary polynomial functions, by proving that for any function f, there exists a multilinear function with the same degree and a recovery threshold no greater than K^*.

For Step 1, we consider the scenario where (1) the domain $\mathbb{V}$ of the function f is in the form of $\mathbb{V} = \mathbb{W}^d$ for some vector space $\mathbb{W}$ and some $d \in \mathbb{N}^+$, and (2) f is a non-zero function of input $X_i = (X_{i,1}, X_{i,2}, ..., X_{i,d}) \in \mathbb{W}^d$, and is multilinear with respect to the elements

$X_{i,1}, X_{i,2}, ..., X_{i,d}$. In this scenario, we develop a lower bound on the minimum recovery threshold as stated in the following lemma.

Lemma 3.6. For any multilinear function f of degree $\deg f \in \mathbb{N}^+$, its minimum recovery threshold is lower bounded by $(K-1)\deg f + 1$ when $N \geq K \deg f - 1$, and lower bounded by $N - \lfloor N/K \rfloor + 1$ when $N < K \deg f - 1$. Moreover, this recovery threshold cannot be further reduced even if we allow arbitrary decoding functions.

We present the proof of Lemma 3.6 in Appendix A. The main idea is to show that for any computing strategy that tries to operate at a recovery threshold smaller than the lower bound stated in Lemma 3.6, there would be scenarios where all available computing results are degenerated (i.e., constants), while the computing results needed by the master are variable, thus violating the decodability requirement.

Next in Step 2, we prove the matching converse for any polynomial function. Given any function f with degree d, we first construct a non-zero, multilinear function f' with the same degree. Then we let $K^*_f(K, N)$ denote the minimum recovery threshold for function f, and prove $K^*_f(K, N) \geq K^*_{f'}(K, N)$, by constructing a computation design of f' that is based on a computation design of f and achieves the same recovery threshold. The construction and its properties are stated in the following lemma, whose proof can be found in Yu *et al.*, 2018c, Appendix E.

Lemma 3.7. Given any function f of degree d, let f' be a map from $\mathbb{V}^d \to \mathbb{U}$ such that $f'(Z_1, ..., Z_d) = \sum_{\mathcal{S} \subseteq [d]} (-1)^{|\mathcal{S}|} f(\sum_{j \in \mathcal{S}} Z_j)$ for any $\{Z_j\}_{j \in [d]} \in \mathbb{V}^d$. Then f' is multilinear with respect to the d inputs. Moreover, if the characteristic of the base field $\mathbb{F}$ is 0 or greater than d, then f' is non-zero.

Given Lemma 3.7, it suffices to prove that f' cannot have a greater recovery threshold than f, i.e. $K^*_f(K, N) \geq K^*_{f'}(K, N)$ for any K and N. We prove this fact by constructing computing schemes for f' given any design for f, which achieve the same recovery threshold.

Note that f' is defined as a linear combination of functions $f(\sum_{j \in \mathcal{S}} Z_j)$, each of which is a composition of a linear map and f.

Given the linearity assumption of the encoding design, any computation scheme of f can be directly applied to any of these functions, achieving the same recovery threshold. Since the decoding functions are linear, the same scheme also applies to linear combinations of them, which includes f'. Hence, the minimum recovery threshold of f' is upper bounded by the recovery threshold of any computing design of f, which indicates $K^*_f(K,N) \geq K^*_{f'}(K,N)$.

To conclude, using the matching converse we proved in Lemma 3.6 for multilinear functions, we showed that the same converse holds in general. This completes the proof of Theorem 3.5.

3.2.5 Application of LCC to accelarate least-squares regression

we demonstrate a practical application of LCC in accelerating distributed least-squares linear regression, whose gradient computation is a quadratic function of the input dataset, hence matching well the LCC framework. We also experimentally demonstrate its performance gain over state-of-the-art straggler mitigation schemes via experiments on AWS EC2 clusters.

Distributed gradient descent for regression problems

We focus on linear regression problems with a least-squares objective. Given a training dataset consisting of m feature inputs $\boldsymbol{x}_i \in \mathbb{R}^d$ and labels $y_i \in \mathbb{R}$ we wish to find the coefficients $\boldsymbol{w} \in \mathbb{R}^d$ of a linear function $\boldsymbol{x} \mapsto \langle \boldsymbol{x}, \boldsymbol{w} \rangle$ that best fits this training data. Minimizing the empirical risk leads to the following optimization problem

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \mathcal{L}(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} (\boldsymbol{x}_i^\top \boldsymbol{w} - y_i)^2 = \frac{1}{m} ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||^2. \tag{3.22}$$

Here, $\boldsymbol{X} = [\boldsymbol{x}_1\ \boldsymbol{x}_2 \cdots\ \boldsymbol{x}_m]^\top \in \mathbb{R}^{m \times d}$ is the feature matrix and $\boldsymbol{y} = [y_1\ y_2 \cdots\ y_m]^\top \in \mathbb{R}^m$ is the output vector obtained by concatenating the input features and output labels, respectively.

Many *nonlinear* regression problems can also be written in the form above. In particular, consider the problem of finding the best function

h belonging to a hypothesis class $\mathcal{H}$ that fits the training data

$$\min_{h\in\mathcal{H}} \mathcal{L}(h) = \frac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}_i) - y_i)^2. \tag{3.23}$$

Such nonlinear regression problems can often be cast in the form (3.22), and be solved efficiently using the so called kernalization trick Schölkopf *et al.*, 2001. However, for simplicity of exposition we focus on the simpler instance (3.22).

A popular approach to solve the above problem is via gradient descent (GD). In particular, GD iteratively refines the weight vector $\boldsymbol{w}$ by moving along the negative gradient direction via the following updates

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta^{(t)}\nabla\mathcal{L}(\boldsymbol{w}^{(t)}) = \boldsymbol{w}^{(t)} - \eta^{(t)}\frac{2}{m}\boldsymbol{X}^\top(\boldsymbol{X}\boldsymbol{w}^{(t)} - \boldsymbol{y}). \tag{3.24}$$

Here, $\eta^{(t)}$ is the learning rate in the tth iteration.

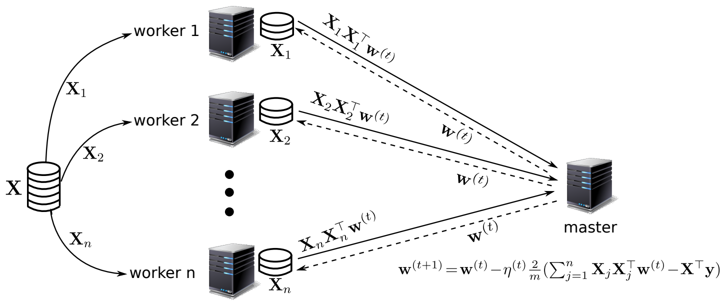

worker 1
worker 2
worker n
$\mathbf{X}$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_n$
$\boldsymbol{X}_1\boldsymbol{X}_1^\top\mathbf{w}(t)$
$\boldsymbol{X}_2\boldsymbol{X}_2^\top\mathbf{w}(t)$
$\boldsymbol{X}_n\boldsymbol{X}_n^\top\mathbf{w}^{(t)}$
$\mathbf{w}(t)$
$\mathbf{w}(t)$
$\mathbf{w}^{(t)}$
master
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta^{(t)}\frac{2}{m}(\sum_{j=1}^{n}\mathbf{X}_j\mathbf{X}_j^\top\mathbf{w}^{(t)} - \mathbf{X}^\top\mathbf{y})$

Figure 3.6: An illustration of a master/worker architecture for data-parallel distributed linear regression.

When the size of the training data is too large to store/process on a single machine, the GD updates can be calculated in a distributed fashion over many computing nodes. As illustrated in Figure 3.6, we consider a computing architecture that consists of a master node and n worker nodes. Using a naive data-parallel distributed regression scheme, we first partition the input data matrix $\boldsymbol{X}$ into n equal-sized sub-matrices such that $\boldsymbol{X} = [\boldsymbol{X}_1 \cdots \boldsymbol{X}_{n-1}]^\top$, where each sub-matrix $\boldsymbol{X}_j \in \mathbb{R}^{d\times\frac{m}{n}}$ contains $\frac{m}{n}$ input data points, and is stored on worker j. Within each iteration of the GD procedure, the master broadcasts the current weight vector $\boldsymbol{w}$ to all the workers. Upon receiving $\boldsymbol{w}$, each worker j computes $\boldsymbol{X}_j \boldsymbol{X}_j^\top \boldsymbol{w}$, and returns it to the master. The master waits for the results from all workers and sums them up to obtain the full gradient

$$\boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{w} = \sum_{j=0}^{n-1} \boldsymbol{X}_j \boldsymbol{X}_j^\top \boldsymbol{w}. \tag{3.25}$$

Then, the master uses this gradient to update the weight vector via (3.24). [4]

Coded computation schemes and their recovery thresholds

The above naive uncoded scheme requires the master to wait for results from all the workers. Therefore, even a single straggler can significantly delay the iteration. One way to combat stragglers is through redundant data storage/processing. For example, each worker, instead of 1, stores and processes $1 < r \leq n$ sub-matrices. Then, we can partition the n sub-matrices into $\frac{n}{r}$ batches of size r, and repeatedly store each batch on r workers. Utilizing this storage/computation redundancy, in the worst case, the master needs the results returned from the fastest $n - r + 1$ workers to compute the final gradient. In general, for a given storage/computation load, we can design optimal coding techniques to minimize the number of workers the master needs to wait for before recovering the gradient. Motivated by this idea, we consider a general distributed regression framework with an input feature matrix $\boldsymbol{X} = [\boldsymbol{X}_1 \cdots \boldsymbol{X}_n]^\top$ and n workers. Each worker j stores r (potentially coded) sub-matrices locally. In each iteration, each worker performs local computation utilizing the received weight vector $\boldsymbol{w}$ and the locally stored data. The master waits for the results from a subset $\mathcal{N} \subseteq [n]$ of workers, and uses them to compute the gradient in (3.25). For this framework, a coded computation scheme consists of the following elements.

- **Computation/storage parameter.** We characterize the computation/storage load at each worker via a parameter $r \in [n]$. Specifically,

[4]Since the value of $\boldsymbol{X}^\top \boldsymbol{y}$ does not vary across iterations, it only needs to be computed once. We assume that it is available at the master for weight updates.

each worker stores some data generated from the feature matrix $\boldsymbol{X}$ whose size is $\frac{r}{n}$-fraction of the size of $\boldsymbol{X}$.

- **Encoding functions.** We encode the data stored at the workers via a set of n encoding functions $\boldsymbol{\rho} = (\rho_1, \ldots, \rho_n)$ where ρ_j is the encoding function of worker j. Each ρ_j maps the input data $\boldsymbol{X}$ into r coded sub-matrices $\tilde{\boldsymbol{X}}_{j,1}, \ldots, \tilde{\boldsymbol{X}}_{j,r} \in \mathbb{R}^{d \times \frac{m}{n}}$ which are locally stored at worker j. In particular, each $\tilde{\boldsymbol{X}}_{j,k}$, is a linear combination of the sub-matrices $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$, i.e.,

$$\tilde{\boldsymbol{X}}_{j,k} = \sum_{i=1}^{n} a_{j,k,i} \boldsymbol{X}_i. \tag{3.26}$$

  Here, the coefficients $a_{j,k,i}$ are specified by the encoding function ρ_j of worker j.

- **Computation functions.** Each worker uses the r encoded sub-matrices along with the weight vector $\boldsymbol{w}$ received from the master to perform its computation. We use $\phi_j : \mathbb{R}^{d \times \frac{m}{n} \times r} \times \mathbb{R}^d \to \mathbb{R}^{\ell_j}$ to denote this mapping whose output is an arbitrary length-ℓ_j vector that is computed locally at worker j using $\tilde{\boldsymbol{X}}_{j,1}, \ldots, \tilde{\boldsymbol{X}}_{j,r}$ and $\boldsymbol{w}$.

- **Decoding function.** The master uses a decoding function $\psi : \underset{j \in \mathcal{N}}{\times} \mathbb{R}^{\ell_j} \to \mathbb{R}^d$ to map the computation results of the available workers in $\mathcal{N}$ to the desired computation $\boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{w}$.

Definition 3.3. We define the recovery threshold of a computation scheme S with a computation/storage load r at each worker, denoted by $K_S(r)$, as the minimum number of workers the master needs to wait to accomplish the gradient computation.

Consider a distributed linear regression task executed on n workers with a local computation/storage load r each. We are interested in finding the minimum recovery threshold achieved among all computation schemes along with the corresponding scheme. This *optimal recovery threshold* can be formally defined as

$$K^*(r) := \min_{S} K_S(r). \tag{3.27}$$

State-of-the-art schemes. Proposed in Tandon *et al.*, 2017, and extended in Halbawi *et al.*, 2017; Raviv *et al.*, 2017; Ye and Abbe, 2018; Li *et al.*, 2018c, the gradient coding (GC) schemes code across partial gradients computed from *uncdoed* data batches to mitigate stragglers for general distributed machine learning problems. In this case, GC schemes achieve a recovery threshold of $K_{\text{GC}}(r) = n - r + 1$. To see this first note that each worker stores r uncoded sub-matrices. For example, using the cyclic repetition scheme in Tandon *et al.*, 2017, worker j stores $\boldsymbol{X}_j, \ldots, \boldsymbol{X}_{j+r-1}$ locally, and sends a liner combination of the computation results $\boldsymbol{X}_j \boldsymbol{X}_j^\top \boldsymbol{w}, \ldots, \boldsymbol{X}_{j+r-1}\boldsymbol{X}_{j+r-1}^\top \boldsymbol{w}$ to the master, who can recover the final result $\boldsymbol{X}_1\boldsymbol{X}_1^\top \boldsymbol{w} + \cdots + \boldsymbol{X}_n\boldsymbol{X}_n^\top \boldsymbol{w}$ by linearly combining the messages received from any subsets of $n - r + 1$ workers.

On the other hand, the matrix-vector multiplication based (MVM) scheme proposed in Lee *et al.*, 2018 takes a different decomposition of the computation $\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w}$ from (3.25). Specifically, the overall computation consists of two rounds. In the first round, an intermediate vector $\boldsymbol{z} = \boldsymbol{X}\boldsymbol{w}$ is computed distributedly and decoded at the master. In the second round, the master re-distributes $\boldsymbol{z}$ to the workers and has them collaboratively compute the final result $\boldsymbol{X}^\top \boldsymbol{z}$. Each worker stores coded data generated using MDS codes from $\boldsymbol{X}$ and $\boldsymbol{X}^\top$ respectively. MVM achieves a recovery threshold of $K_{\text{MVM}}(r) = \lceil \frac{2n}{r} \rceil$ in *each* round, when the storage is evenly split between rounds. It was recently proposed in Maity *et al.*, 2018 to use one round of matrix-vector multiplication to compute the gradient, given that the second moment of the feature matrix $\boldsymbol{X}^\top \boldsymbol{X}$ is known in prior. However, since we focus on the cases where the input $\boldsymbol{X}$ is very large, storing $\boldsymbol{X}$ and computing $\boldsymbol{X}^\top \boldsymbol{X}$ on a single machine is prohibitive.

Applying LCC to minimize recovery threshold

We note that above gradient computation framework can be cast to the computation model in Section 3.2.1. To do that, we group the sub-matrices into $K = \lceil \frac{n}{r} \rceil$ data blocks such that $\boldsymbol{X} = [\bar{\boldsymbol{X}}_1 \cdots \bar{\boldsymbol{X}}_K]^\top$. Then the gradient computation (3.25) reduces to computing the sum of a degree-2 polynomial $f(\bar{\boldsymbol{X}}_k) = \bar{\boldsymbol{X}}_k \bar{\boldsymbol{X}}_k^\top \boldsymbol{w}$, evaluated over K data blocks $\bar{\boldsymbol{X}}_1, \ldots, \bar{\boldsymbol{X}}_K$.

Now, we can directly apply LCC to minimize the recovery threshold in each iteration. We first generate the coded matrix $\tilde{\boldsymbol{X}}_i$ stored at worker i as a linear combination of $\bar{\boldsymbol{X}}_1, \ldots, \bar{\boldsymbol{X}}_K$ as in (3.21). Each worker i computes $f(\tilde{\boldsymbol{X}}_i) = \tilde{\boldsymbol{X}}_i \tilde{\boldsymbol{X}}_i^\top \boldsymbol{w}$, and sends it to the master. Based on Theorem 3.5 the master can recover $f(\bar{\boldsymbol{X}}_1), \ldots, f(\bar{\boldsymbol{X}}_K)$ and the gradient by summing them up after receiving the results from $2(K-1)+1 = 2\lceil\frac{n}{r}\rceil - 1$ workers. We state this result in the following corollary.

Corollary 3.8. Consider the above distributed linear regression problem and assume it is executed over n workers, each storing $2 \le r \le n$ coded sub-matrices. In this setting, LCC achieves a recovery threshold of $K_{\text{LCC}}(r) = 2\lceil\frac{n}{r}\rceil - 1$. Furthermore, the recovery threshold achieved by LCC is within a factor two of the minimum possible recovery threshold $K^*(r)$ achievable by any algorithm. That is

$$\frac{1}{2}K_{\text{LCC}}(r) < K^*(r) \le K_{\text{LCC}}(r) = 2\lceil\tfrac{n}{r}\rceil - 1, \tag{3.28}$$

When $r = 1$, LCC reduces to the uncoded scheme where each worker j computes $\boldsymbol{X}_j \boldsymbol{X}_j^\top \boldsymbol{w}$. The achievability part directly comes from the recovery threshold of LCC. As for the converse part, Since here we consider a more general scenario where workers can execute any computation on the data (not necessarily matrix-matrix multiplication), the lower bound in Theorem 3.5 no longer holds. we refer the interested readers to Li *et al.*, 2018d for the proof of a new lower bound on $K^*(r)$ that is no less than half of $K_{\text{LCC}}(r)$.

Remark 3.13. We note that LCC is also directly applicable for non-linear regression problems using kernel methods. To do that, we simply replace the data matrix $\boldsymbol{X}$ with the kernel matrix $\boldsymbol{\mathcal{K}}$, whose entry $\mathcal{K}_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is some kernel function of the data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$.

Comparison with state of the arts. Compared with the gradient coding (GC) schemes (see, e.g., Tandon *et al.*, 2017; Halbawi *et al.*, 2017; Raviv *et al.*, 2017), LCC directly codes across the raw data before computation, further reducing the recovery threshold by about $r/2$ times. While the amount of computation and communication at each worker is the same for GC and LCC, LCC is expected to finish much faster due to its much smaller recovery threshold. However, GC schemes are applicable for more general learning problems where the gradient can be arbitrary functions of the data.

Compared with the matrix-vector multiplication based (MVM) scheme in Lee *et al.*, 2018, LCC completes each iteration in only one round of computation and communication, with a smaller recovery threshold than that of MVM in each round (assuming even storage split between two rounds). However, MVM requires less amount of computation at each worker than LCC. While LCC has each worker send a dimension-d vector in each iteration, each MVM worker sends two vectors whose sizes are respectively proportional to m and d.

Experiments on AWS EC2

We run distributed linear regression on Amazon EC2 clusters, and empirically compare the performance of the proposed LCC scheme with the conventional uncoded scheme for which there is no data redundancy among the workers, the GC scheme (specifically, the cyclic repetition scheme in Tandon *et al.*, 2017), and the MVM scheme in Lee *et al.*, 2018.

Setup. We train a linear regression model using Nesterov's accelerated gradient descent over a distributed computing system, where the master and worker nodes are implemented on `t2.micro` instances using `Python`. Message passing between instances are impelmendted using `MPI4py` Dalcin *et al.*, 2011. In each iteration, each worker sends its computation result back to the master asynchronously using `Isend()`.

Data. We create synthetic datasets of m training samples by 1) sampling a true weight vector $\boldsymbol{w}^*$ whose components are i.i.d. and uniformly distributed on $[0, 1]$, and 2) sampling each input point $\boldsymbol{x}_i$ of d features from a normal mixture distribution $\frac{1}{2} \times \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{I}) + \frac{1}{2} \times \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{I})$, where $\boldsymbol{\mu}_1 = \frac{1.5}{d}\boldsymbol{w}^*$ and $\boldsymbol{\mu}_2 = \frac{-1.5}{d}\boldsymbol{w}^*$, and computing its output label $y_i = \boldsymbol{x}_i^\top \boldsymbol{w}^*$. For each dataset, we run GD for 100 iterations over $n = 40$ workers. We consider different dimensions of input matrix $\boldsymbol{X}$ as listed in the following scenarios.

- Scenario 1 & 2: $(m, d) = (8000, 7000)$.

- Scenario 3: $(m, d) = (160000, 500)$.

We let the system run with naturally occurring stragglers in scenario 1. To mimic the effect of slow/failed workers, we artificially introduce stragglers in scenarios 2 and 3, by imposing a 0.5 seconds delay on each worker with probability 5% in each iteration.

To implement LCC, we set the β_i parameters to $1, ..., \frac{n}{r}$, and the α_i parameters to $0, \ldots, n-1$. To avoid numerical instability due to large entries of the decoding matrix, we can embed input data into a large finite field, and apply LCC in it with exact computations. However in all of our experiments the gradients are calculated correctly without carrying out this step.

Results. For the uncoded scheme, each worker stores and processes $r = 1$ data batch. For the GC and LCC schemes, we select the optimal r subject to the memory size of the `t2.micro` instance to minimize the total run-time. For MVM, we further optimized the run-time over the computation/storage assigned between two rounds of matrix-vector multiplications. We plot the run-time performance in all three scenarios in Figure 3.7, and also list the breakdowns of their run-times in Tables 3.1 to 3.3. The computation time was measured as the summation of the maximum local processing time among all non-straggling workers, over 100 iterations. The communication time is computed as the difference between the total run-time and the computation time.

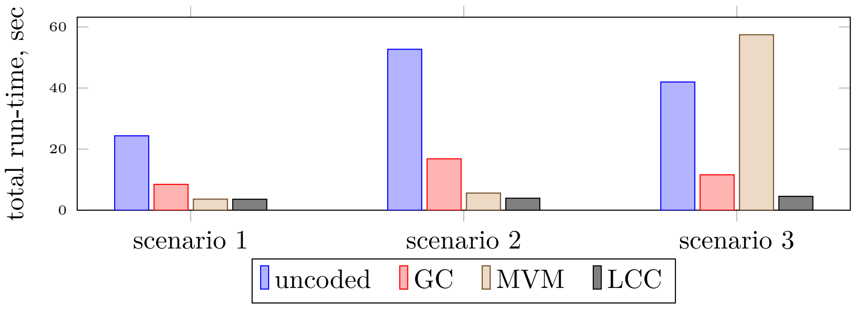


Figure 3.7: Run-time comparison of LCC with other three schemes: conventional uncoded, GC, and MVM.

Based on the experimental results, we draw the following conclusions.

- LCC achieves the least run-time in all scenarios. In particular, LCC

Table 3.1: Breakdowns of the run-times in scenario one.

schemes	# batches/ worker (r)	recovery threshold	communication time	computation time	total run-time
uncoded	1	40	24.125 s	0.237 s	24.362 s
GC	10	31	6.033 s	2.431 s	8.464 s
MVM Rd. 1	5	8	1.245 s	0.561 s	1.806 s
MVM Rd. 2	5	8	1.340 s	0.480 s	1.820 s
MVM total	10	-	2.585 s	1.041 s	3.626 s
LCC	10	7	1.719 s	1.868 s	3.587 s

Table 3.2: Breakdowns of the run-times in scenario two.

schemes	# batches/ worker (r)	recovery threshold	communication time	computation time	total run-time
uncoded	1	40	7.928 s	44.772 s	52.700 s
GC	10	31	14.42 s	2.401 s	16.821 s
MVM Rd. 1	5	8	2.254 s	0.475 s	2.729 s
MVM Rd. 2	5	8	2.292 s	0.586 s	2.878 s
MVM total	10	-	4.546 s	1.061 s	5.607 s
LCC	10	7	2.019 s	1.906 s	3.925 s

Table 3.3: Breakdowns of the run-times in scenario three.

schemes	# batches/ worker (r)	recovery threshold	communication time	computation time	total run-time
uncoded	1	40	0.229 s	41.765 s	41.994 s
GC	10	31	8.627 s	2.962 s	11.589 s
MVM Rd. 1	5	8	3.807 s	0.664 s	4.471 s
MVM Rd. 2	5	8	52.232 s	0.754 s	52.986 s
MVM total	10	-	56.039 s	1.418 s	57.457 s
LCC	10	7	1.962 s	2.597 s	4.541 s

speeds up the uncoded scheme by 6.79×-13.43×, the GC scheme by 2.36-4.29×, and the MVM scheme by 1.01-12.65×.

- In scenarios 1 & 2 where the number of inputs m is close to the number of features d, LCC achieves a similar performance as MVM. However, when we have much more data points in scenario 3, LCC finishes substantially faster than MVM by as much as 12.65×. The main reason for this subpar performance is that MVM requires large amounts of data transfer from workers to the master in the first round and from master to workers in the second round (both are proportional to m). However, the amount of communication from each worker or master is proportional to d for all other schemes, which is much smaller than m in scenario 3.

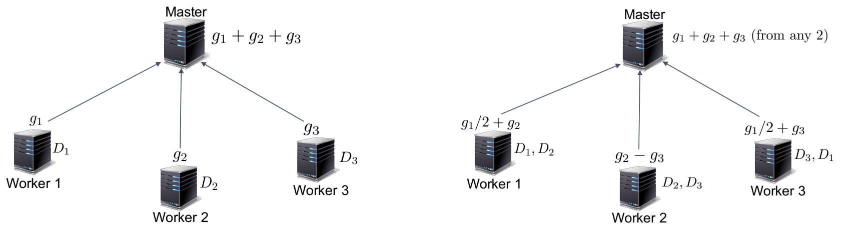
3.3 Related works and open problems

Unified coding. So far, we have demonstrated how coded computing techniques can inject and leverage redundant computations to minimize the load of communication, and the effect of stragglers, respectively. Moving beyond these individual improvements, we have recently proposed in Li *et al.*, 2016a a *unified* coded framework for distributed computing with straggling servers, by introducing a tradeoff between "latency of computation" and "load of communication" for some linear computation tasks. We showed that the Coded Distributed Computing (CDC) scheme in Chapter 2 that repeats the intermediate computations to create coded multicasting opportunities to reduce communication load, and the coded scheme of Lee *et al.*, 2018 that generates redundant intermediate computations to combat straggling servers can be viewed as special instances of the proposed framework, by considering two extremes of this tradeoff: minimizing either the load of communication or the latency of computation individually. The key idea of this unified coding scheme is to apply redundant CDC data placement on MDS-coded data blocks. Then, by tuning the coding rate of the MDS code and the number of redundant computations for each intermediate task, we can systematically operate at any point on the latency-load tradeoff to optimize the run-time performance of distributed computing tasks. We also proved an information-theoretic lower bound on the latency-load tradeoff, which was shown to be within a constant multiplicative gap from the achieved tradeoff at the two end points.

Gradient coding. While coded computing schemes have been designed to speed up fundamental algebraic computations like matrix multiplication and polynomial evaluation, directly adopting these schemes in general machine learning algorithms is often not applicable since the gradient computation may not have any algebraic structure, or can only be evaluated numerically. One of the most important learning algorithms is the stochastic gradient descent (SGD), which is currently the most widely used training method in supervised learning. Recently, a coding method named "gradient coding" (GC) was proposed in Tandon *et al.*, 2017, and extended in Halbawi *et al.*, 2017; Raviv *et al.*, 2017; Ye and Abbe, 2018, to mitigate stragglers in running distributed SGD. We use the following simple example to illustrate the idea of GC.

Figure 3.8(a) illustrates a naive way of distributing the computation of the gradient on three workers. The three workers have disjoint partitions of the data stored locally (D_1,D_2,D_3) and all share the current model. For $i = 1, 2, 3$, Worker i computes the gradient of the model on examples in partition D_i, denoted by g_i. The three gradient vectors are then communicated to a master node which computes the full gradient by summing these vectors $g_1 + g_2 + g_3$ and updates the model with a gradient step. The new model is then sent to the workers and the system moves to the next iteration. This setup is, however, subject to delays introduced by stragglers because the master has to wait for outputs of all three workers before computing $g_1 + g_2 + g_3$.

Figure 3.8(b) illustrates one way to resolve this problem by replicating data across machines as shown, and sending linear combinations of the associated gradients. As shown in Figure 3.8(b), each data partition is replicated twice using a specific placement policy. Each worker is assigned to compute two gradients on their assigned two data partitions. For instance, Worker 1 computes vectors g_1 and g_2, and then sends $\frac{1}{2}g_1+g_2$. Interestingly, $g_1+g_2+g_3$ can be constructed from *any two out of these three vectors.* For instance, $g_1 + g_2 + g_3 = 2\left(\frac{1}{2}g_1 + g_2\right) - (g_2 - g_3)$. Therefore, such a scheme is robust to one straggler. This gradient cod-



(a) Naive synchronous gradient descent. (b) Gradient coding: The vector $g_1 + g_2 + g_3$ is in the span of *any two*.

Figure 3.8: Illustration of gradient coding.

ing technique makes the computation robust to stragglers albeit at a computational overhead compared to the naive scheme, while keeping the communication load the same.

In general GC schemes require processing $s + 1$ data batches at each worker in order for the system to tolerate s stragglers. We also note that in contrast to the previously proposed Lagrange Coded Computing scheme that codes over data batches, the GC schemes code over partial gradients computed from uncoded data, hence it is applicable to arbitrary loss functions whose gradients may not have any algebraic structure or can only be computed numerically (e.g., deep neural networks).

Finally, we end this chapter with some of the open problems and future directions for designing straggler-resilient coded computing systems.

Low-complexity algorithms for coded matrix multiplication. While the naive multiplication of an $M \times N$ matrix $\mathbf{A}$ by an $N \times L$ matrix $\mathbf{B}$ has complexity $O(MNL)$, there is a rich literature that has discovered low complexity implementations, especially if the matrices are restricted to a certain class. When the entries of matrix $\mathbf{A}$ come from a *bounded alphabet* $\mathcal{A}$ (e.g., $\mathbf{A}$ is the adjacency matrix of a degree-bounded graph in common graph algorithms like pagerank, or Laplacian matrix calculation), the product $\mathbf{AB}$ can be computed via *the four Russians algorithm* Ullman *et al.*, 1974; Liberty and Zucker, 2009 using

$O(MNL\log_2|\mathcal{A}|/\log_2 N)$ operations - an improvement of a factor of $\log_2 N$ as compared to the naive approach for small alphabet. There are some unique challenges for the use of the four Russians method in coded distributed matrix multiplication due to the fact that the alphabet size of good codes tends to be large. Consider a concrete example where $\mathcal{A}=\{0,1\}$, and $\mathbf{B}$ is a $N\times 1$ vector. Surprisingly, a back-of-the envelop calculation reveals that natural application of MDS codes to the case of binary multiplication *has the same per-node computational complexity as replication* $O(\frac{MN(s+1)}{n\log_2 N})$, for a fixed straggler tolerance s. This is because the alphabet size of a parity matrix, say $\sum_{i=1}^{m} g_i\mathbf{A}_i$, can be as large as 2^m, so the computational complexity of multiplying with $\mathbf{B}$ is $O(\frac{MN}{\log_2 N})$, whereas an uncoded computation has complexity $O(\frac{MN}{m\log_2 N})$. Motivated by this observation, we propose to answer the following question: *For a matrix multiplication where the matrix entries come from a bounded alphabet $\mathcal{A}$, what is the the optimal trade-off between straggler tolerance and the per-node computational complexity.* There have been several recent works along this direction in Haddadpour and Cadambe, 2018; Tang *et al.*, 2019.

Developing "master-less" systems for efficient and straggler-resilient matrix multiplication. Current state of the art in coded computing largely assumes availability of master/fusion nodes that distribute and collect data, and perform encoding/decoding computations. In practice, however, often all nodes are identical, and no single node may be able to store all the data or perform all the encoding/decoding operations (see e.g., Jeong *et al.*, 2018). Distributed and parallel computing literature has developed efficient matrix multiplication algorithms for decentralized architectures, for example, the Scalable Universal Matrix Multiplication Algorithm (SUMMA) Van De Geijn and Watts, 1997, which is implemented in linear algebra libraries such as ScaLAPACK Blackford *et al.*, 1997, PLAPACK Alpatov *et al.*, 1997, PB-BLAS Choi *et al.*, 1996, and Elemental Poulson *et al.*, 2013. A significant challenge that SUMMA overcomes is to limit the communication cost in decentralized architectures. The theoretical basis for practical algorithms like SUMMA comes from the study of their completion time over a distributed computing model Demmel *et al.*, 2012; Ballard *et al.*, 2014 of n fully connected nodes, where message transmission time involves a fixed start up time plus a time that is proportional to the length of the message. The completion time for matrix multiplication in SUMMA is approximately optimal in this model Ballard *et al.*, 2014. The main idea of SUMMA is that it cleverly schedules the operations performed by the nodes to minimize the amount of time spent waiting for data and message communication and startup costs, minimizing the overall completion time. SUMMA, however, is not robust to stragglers or failures. As such motivated, we propose to solve the following problem: *Develop a matrix multiplication algorithm over* n *nodes that is robust to* s *stragglers, and minimizes the completion time in the model of Ballard* et al.*, 2014.* In order to emulate the fusion node's decode/repair functionality over the master-less decentralized systems, one may refer to related designs for locally recoverable and regenerating codes in the distributed storage literature. Also, it would be interesting to try to develop lower bounds on the completion time using the techniques for proving straggler-aware bounds in Dutta *et al.*, 2016; Yu *et al.*, 2017b.

Gradient coding for partial stragglers. Previous works of gradient coding for distributed learning relied on a simplified assumption: straggling machines perform no work *i.e.* fail catastrophically or simply do not respond to requests. In reality, this rarely happens: machines are simply slower because of an OS update, moving of virtualization resources across servers or other issues relating to pooled computing resources. Furthermore a machine may be a straggler for some iterations of the learning process but not for others. This allows us to design methods for iterative learning not one round at a time, but considering the iterative nature of the whole training process jointly.

One way to tackle this problem is to use a layering of different gradient codes designed for different numbers of stragglers. Each data batch of gradient descent can be partitioned into smaller partitions and combinations of gradient codes can provide good intermediate performance. A good way to explain our future vision for this problem is through erasure codes: In classic MDS erasure coding k data blocks are encoded into n blocks with the guarantee that if someone collects *any* k from the encoded blocks they can reconstruct all the original data.

However, even if $k-1$ blocks are recovered, there is no guarantee of recovery. Of course, one could make a systematic MDS code and have some intermediate performance from the systematic blocks, but it is highly nontrivial to improve on that. For example, one could ask that any $k/2$ blocks suffice to recover a good fraction of the original data and also recover everything from any k blocks.

This problem is sometimes called *intermediate performance* for erasure codes, see e.g., Sanghavi, 2007; Kim and Lee, 2009; Dimakis *et al.*, 2007 and the related growth codes Kamra *et al.*, 2006. The problem we are proposing here is intermediate performance for gradient codes: For example, create a code to ensure that a full gradient is recovered if any $n-s-\ell_1$ machines each process $2k_1$ blocks, and ℓ_1 partial stragglers process some k_2 blocks. In this example we assumed a slowdown factor of 2. Finding the fundamental limits and designing optimal gradient codes for such systems are interesting research problems. Some progress has been made on this direction in Ferdinand and Draper, 2018; Narra *et al.*, 2019.

Gradient coding that produce approximate gradients. Another way to alleviate the computational overhead of gradient coding can be by relaxing the requirement of exactly recovering the full gradient (or of any batch involved in a particular iteration). In other words, one could try to design n sparse vectors $\{b_1, b_2, \ldots, b_n\}$ such that the span of any $(n-s)$ contains a vector *close* to the all 1s vector $\mathbf{1}$. This gives rise to the following open problem. Thinking of the n vectors as rows of a matrix B, and given some constant $\epsilon > 0$, one form of stating this problem requires constructing the matrix B such that any submatrix of $(n-s)$ rows, say B', satisfies $\|B'x - \mathbf{1}\|_2 \leq \epsilon$ for some vector x. This question has been recently studied in Raviv *et al.*, 2017; Charles *et al.*, 2017; Wang *et al.*, 2019b; Wang *et al.*, 2019a. However, the question of whether these approaches are optimal is still open. Furthermore, the problem stated as such is agnostic to the actual gradient being approximated. It is conceivable that building data dependent encoders that exploit gradients from the previous iteration could lead to better approximations of the current gradient for the same computational cost.

Beyond polynomial computations. Polynomial computation is the most general class of computations for which we know the optimal design for coded computing via Lagrange coding. Extending the state-of-the-art in coded computing to go beyond polynomial computations is a very important and challenging research direction, which is expected to impact various application domains (in particular, machine learning, in which non-linear threshold functions are common). There have been some work on this direction in Kosaian *et al.*, 2018; Dutta *et al.*, 2019; Yang *et al.*, 2017; So *et al.*, 2019, however the problem still remains largely unsolved.

4

Coding for Security and Privacy

In the previous chapters, we have demonstrated the role of coding in reducing the bandwidth requirement, and alleviating the stragglers' delay, for distributed computing applications. In this chapter, we focus on addressing another two major concerns of the information age - security and privacy. The security concern in distributed computation is having Byzantine (or malicious) workers with no computational restriction, who can deliberately send erroneous data to affect the computation for their benefit. Examples for such scenarios include the one described in Blanchard *et al.*, 2017b, where it is shown that a single malicious server in a distributed execution of gradient descent can cause arbitrary bias in the resulting hypothesis. In addition to security challenges, distributed computation and learning schemes are susceptible to privacy infringement. Since such computations are commonly performed by using third party cloud services, the concern for personal data leakage is growing. Therefore, in some cases it is crucial to keep the workers oblivious to the actual data they are processing.

Security and privacy have been the main research focus in the literature of *multiparty computing* (MPC) and secure/private machine learning (see, *e.g.*, Ben-Or *et al.*, 1988; Cramer *et al.*, 2001; Halpern and Teague, 2004; Cramer *et al.*, 2015; Mohassel and Zhang, 2017). In this chapter, we demonstrate how coding theory can help to maintain security and privacy in multiparty computing and distributed learning. Specifically, we first demonstrate that how we can extend the Lagrange Coded Computing framework proposed in the previous chapter to provide MPC systems with security and privacy guarantees. We also compare LCC with state-of-the-art MPC schemes (e.g., the celebrated BGW scheme for secure/private MPC Ben-Or *et al.*, 1988), and illustrate the substantial reduction in the amount of randomness, storage overhead, and computational complexity achieved by LCC.

Second, we demonstrate the application of coded computing for privacy-preserving machine learning. In particular, we consider an application scenario in which a data-owner (e.g., a hospital) wishes to train a logistic regression model by offloading the large volume of data (e.g., healthcare records) and computationally-intensive training tasks (e.g., gradient computations) to N machines over a cloud platform, while ensuring that any collusions between T out of N workers do not leak information about the dataset. We then discus a recently proposed scheme So *et al.*, 2019 that leverages coded computing for this problem. We finally end this chapter with a discussion on some related works and open problems.

4.1 Secure and private multiparty computing

We consider the problem of evaluating a multivariate polynomial $f : \mathbb{V} \to \mathbb{U}$ over a dataset $X = (X_1, \ldots, X_K)$, where $\mathbb{V}$ and $\mathbb{U}$ are vector spaces of dimensions M and L, respectively, over the finite field[1] $\mathbb{F}_q$. We assume a distributed computing environment with a master and N workers (see Figure 4.1), and the goal is to compute $Y_1 \triangleq f(X_1), \ldots, Y_K \triangleq f(X_K)$ given function f. We define the degree of the chosen f, denoted by $\deg f$, as the total degree of the polynomial.

In this setting each worker has already stored a fraction of the dataset prior to computation, in a possibly coded manner. Specifically, for $i \in$

[1]While the results about security hold for any large enough field, privacy is well defined only over finite ones.

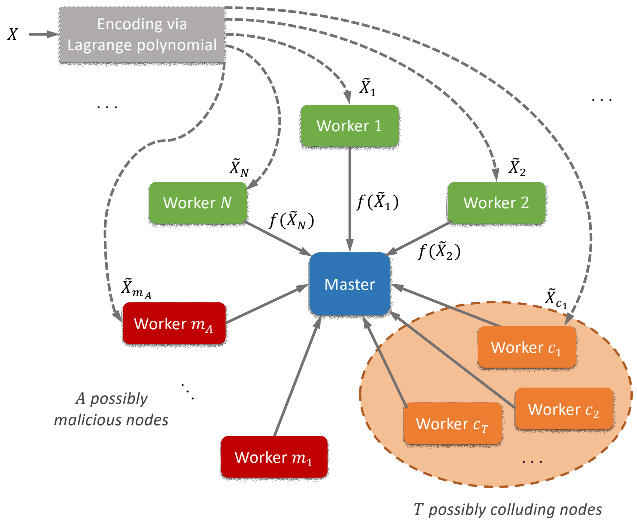


Figure 4.1: An illustration of coded computing in the presence of malicious workers $(m_1, \ldots, m_A)$ who wish to affect the computation for their own benefit, and sets of colluding workers $(c_1, \ldots, c_T)$ who wish to know the dataset X. The master encodes the dataset $\{X_j\}_{j=1}^{K}$ to $\{\tilde{X}_i\}_{i=1}^{N}$, and sends $\tilde{X}_i$ to worker i. In turn, the workers compute $f(\tilde{X}_i)$ and send the result back to the master. The master needs to retrieve $\{f(X_i)\}_{i=1}^{K}$ in the presence of at most A malicious workers, and maintain the perfect privacy of the dataset in the face of up to T colluding workers.

$[N]$ (where $[N] \triangleq \{1, \dots, N\}$), worker i stores $\tilde{X}_i \triangleq g_i(X_1, \dots, X_K, Z)$, where g_i is the encoding function of that worker, and Z is a random variable. We restrict our attention to linear encoding functions, which guarantee low encoding complexity and simple implementation. Specifically, each $\tilde{X}_i$ is a linear combination of $X_1, \dots, X_K, Z$.

Upon starting computation, each worker $i \in [N]$ computes $\tilde{Y}_i \triangleq f(\tilde{X}_i)$ and returns the result to the master. The master waits for all workers and then decodes outputs $Y_1, \dots, Y_K$ using a decoding function given these results.

The procedure described above must satisfy two additional requirements. First, the workers must remain oblivious to the content of the dataset, even if up to T of them collude, where T is the *privacy parameter* of the system. Formally, for every $\mathcal{T} \subseteq [N]$ of size at most T, we must have that[2]

$$I(X; \tilde{X}_{\mathcal{T}}) = 0 \tag{4.1}$$

where I is mutual information, $\tilde{X}_{\mathcal{T}}$ represents the encoded dataset that is stored at the workers in $\mathcal{T}$, and X is seen as chosen uniformly at random. A scheme which guarantees privacy against T colluding workers is called *T-private.*

In addition to privacy, the computing scheme should provide *security*, *i.e.*, robustness against malicious workers. Formally, the master must be able to obtain true values of $Y_1, \dots, Y_K$ even if up to A workers return arbitrarily erroneous results, where A is the *security parameter* of the system. A scheme that guarantees security against A malicious workers is called *A-secure.*

Uncoded repetition scheme. For this setting, a naive uncoded scheme simply replicates each uncoded data block X_i onto multiple workers. By replicating each X_i between $\lfloor N/K \rfloor$ and $\lceil N/K \rceil$ times, it can tolerate at most A adversaries when $2A \leq \lfloor N/K \rfloor - 1$. However, uncoded repetition does not support the privacy requirement.

[2]Equivalently, equation (4.1) requires that $\tilde{X}_{\mathcal{T}}$ and X are independent. Under this condition, the input data X still appears uniformly random after the colluding workers learn $\tilde{X}_{\mathcal{T}}$, which guarantees the privacy.

4.1.1 LCC for secure and private multiparty computing

We star with an illustrative example of how LCC scheme that we described in the previous chapter can be utilized for secure and private MPC.

Illustrative Example

Consider the function $f(X_i) = X_i^2$, where input X_i's are $\sqrt{M} \times \sqrt{M}$ square matrices for some square integer M. We demonstrate LCC in the scenario where the input data X is partitioned into $K = 2$ batches X_1 and X_2, and the computing system has $N = 7$ workers. In addition, the scheme guarantees perfect privacy against any individual worker (i.e., $T = 1$), and is robust against any single malicious worker (i.e., $A = 1$).

The gist of LCC is picking a uniformly random matrix Z of the same dimensions as the X_i's, and to encode (X_1, X_2, Z) using a Lagrange interpolation polynomial u. To this end, assume that the underlying field $\mathbb{F}_q = \mathbb{F}_{11}$, let

$$u(z) \triangleq X_1 \cdot \frac{(z-2)(z-3)}{(1-2)(1-3)} + X_2 \cdot \frac{(z-1)(z-3)}{(2-1)(2-3)} + \\ Z \cdot \frac{(z-1)(z-2)}{(3-1)(3-2)},$$

and observe that $u(1) = X_1$ and $u(2) = X_2$. Then, fix distinct $\{\alpha_i\}_{i=1}^7$ in $\mathbb{F}_{11}$ such that $\{\alpha_i\}_{i=1}^7 \cap [2] = \varnothing$, and have workers $1, \ldots, 7$ store $u(\alpha_1), \ldots, u(\alpha_7)$, i.e.,

$$\left(\tilde{X}_1, \ldots, \tilde{X}_7\right) = (X_1, X_2, Z) \cdot U$$

where $U \in \mathbb{F}_{11}^{3\times 7}$ satisfies $U_{i,j} = \prod_{\ell \in [3]\setminus\{i\}} \frac{\alpha_j - \ell}{i - \ell}$ for $(i,j) \in [3] \times [7]$.

First, notice that for every $j \in [7]$, worker j sees $\tilde{X}_j$, which is a linear combination of X_1 and X_2 masked by addition of $\lambda \cdot Z$ for some nonzero $\lambda \in \mathbb{F}_{11}$; since Z is uniformly random, this guarantees perfect privacy for $T = 1$. Next, worker j computes $f(\tilde{X}_j) = f(u(\alpha_j))$, which is an evaluation of the composition polynomial $f(u(z))$, with degree at most 4, at α_j.

Normally, a polynomial of degree 4 can be interpolated from 5 evaluations at distinct points. However, the presence of $A = 1$ malicious worker requires the master to employ a Reed-Solomon decoder, and have two additional evaluations at distinct points (in general, two additional evaluations for every malicious worker). Finally, after decoding polynomial $f(u(z))$, the master can obtain $f(X_1)$ and $f(X_2)$ by evaluating it at $z = 1$ and $z = 2$.

General Description

To start, we first select any $K+T$ distinct elements $\beta_1, \dots, \beta_{K+T}$ from $\mathbb{F}$, and find a polynomial $u : \mathbb{F} \to \mathbb{V}$ of degree $K+T-1$ such that $u(\beta_i) = X_i$ for any $i \in [K]$, and $u(\beta_i) = Z_i$ for $i \in \{K+1, \dots, K+T\}$, where all Z_i's are chosen uniformly at random from $\mathbb{V}$. This is accomplished by letting u be the respective Lagrange interpolation polynomial

$$u(z) \triangleq \sum_{j\in[K]} X_j \cdot \prod_{k\in[K+T]\setminus\{j\}} \frac{z-\beta_k}{\beta_j-\beta_k} + \sum_{j=K+1}^{K+T} Z_j \cdot \prod_{k\in[K+T]\setminus\{j\}} \frac{z-\beta_k}{\beta_j-\beta_k}. \tag{4.2}$$

We then select N distinct elements $\alpha_1, \dots, \alpha_N$ from $\mathbb{F}$ such that $\{\alpha_i\}_{i=1}^N \cap \{\beta_j\}_{j=1}^K = \varnothing$, and encode the input variables by letting $\tilde{X}_i = u(\alpha_i)$ for any $i \in [N]$. That is, the input variables are encoded as

$$\tilde{X}_i = u(\alpha_i) = (X_1, \dots, X_K, Z_{K+1}, \dots, Z_{K+T}) \cdot U_i, \tag{4.3}$$

where $U \in \mathbb{F}_q^{(K+T)\times N}$ is the encoding matrix $U_{i,j} \triangleq \prod_{\ell\in[K+T]\setminus\{i\}} \frac{\alpha_j-\beta_\ell}{\beta_i-\beta_\ell}$, and U_i is its i'th column.

Next we briefly sketch the proof of T-privacy, which relies on the fact that the bottom $T \times N$ submatrix U^{bottom} of U is an MDS matrix (i.e., every $T \times T$ submatrix of U^{bottom} is invertible). Hence, for a colluding set of workers $\mathcal{T} \subseteq [N]$ of size T, their encoded data $\tilde{X}_{\mathcal{T}}$ satisfies $\tilde{X}_{\mathcal{T}} = XU_{\mathcal{T}}^{top} + ZU_{\mathcal{T}}^{bottom}$, where $Z \triangleq (Z_{K+1}, \dots, Z_{K+T})$, and $U_{\mathcal{T}}^{top} \in \mathbb{F}_q^{K\times T}$, $U_{\mathcal{T}}^{bottom} \in \mathbb{F}_q^{T\times T}$ are the top and bottom sub-matrices which correspond to the columns in U that are indexed by $\mathcal{T}$.

Now, the fact that U^{bottom} is MDS implies that $U^{bottom}_{\mathcal{T}}$ is invertible, and hence

$$Z = (\tilde{X}_{\mathcal{T}} - XU^{top}_{\mathcal{T}}) \cdot (U^{bottom}_{\mathcal{T}})^{-1}.$$

Therefore, for *every* dataset X and *every* observed encoding $\tilde{X}_{\mathcal{T}}$, there exists a unique value for the randomness Z by which the encoding of X equals $\tilde{X}_{\mathcal{T}}$; a statement equivalent to the definition of T-privacy.

Following the encoding of (4.3), each worker i applies f on $\tilde{X}_i$ and sends the result back to the master. Hence, the master obtains N evaluations, at most A of which are incorrect, of the polynomial $f(u(z))$. Since $\deg f(u(z)) \leq \deg f \cdot (K+T-1)$, and $N \geq (K+T-1)\deg(f) + 2A+1$, the master can obtain all coefficients of $f(u(z))$ by applying Reed-Solomon decoding. Having this polynomial, the master evaluates it at β_i for every $i \in [K]$ to obtain $f(u(\beta_i)) = f(X_i)$. This results in the following theorem for LCC.

Theorem 4.1. Given a number of workers N and a dataset $X = (X_1, \ldots, X_K)$, LCC scheme provides an A-secure, and T-private computation of $\{f(X_i)\}_{i=1}^{K}$ for any polynomial f, as long as

$$(K+T-1)\deg f + 2A + 1 \leq N, \tag{4.4}$$

for any field $\mathbb{F}_q$ that is sufficiently large (i.e., $q \geq N+K$).

Remark 4.1. This construction is applicable over every finite field with $q \geq K+N$. Moreover, disregarding the privacy constraint (i.e., setting $T=0$) provides an A-secure scheme over infinite fields as well.

Remark 4.2. Note that LHS of inequality (4.4) is independent of the number of workers N, hence the key property of LCC is that adding 1 worker can increase its security to malicious workers by 1/2, while keeping the privacy constraint T the same. This result essentially extends the well-known optimal scaling of error-correcting codes (i.e., adding one parity can provide robustness against one erasure or 1/2 error in optimal maximum distance separable codes) to the distributed computing paradigm.

4.1.2 Optimality of LCC for secure and private MPC

We now discuss the optimality of LCC by proving Theorem 4.2 (optimal security) and Theorem 4.3 (optimal randomness), stated below.

Theorem 4.2 (Optimal security). For any multilinear function f, security can be provided against at most $A = \lfloor (N - (K-1)\deg\ f - 1)/2 \rfloor$ adversaries when $N \geq K\deg\ f - 1$, and $A = \lfloor N/2K - 1/2 \rfloor$ adversaries when $N < K\deg\ f - 1$.

Compared with the result in Theorem 4.1 (for the case of $T = 0$), Theorem 4.2 demonstrates that the LCC scheme provides the optimal security, by protecting against maximum possible number of adversaries.

Proof of Theorem 4.2

We prove Theorem 4.2 by connecting the adversary tolerance problem to the straggler mitigation problem described in Section 3.2.1, using the extended concept of Hamming distance for coded computing.

As the first step, we define the Hamming distance of a (possibly random) linear encoding scheme, denoted by d, as the maximum integer, such that for any two distinct instances of input X that also generate distinct outputs, and for any two possible realizations of the N encoding functions, the encoded data differ for at least d workers.

It was shown in Yu *et al.*, 2018a that this Hamming distance behaves similar to its counterpart in classical coding theory: an encoding scheme can tolerate S stragglers and A adversarial workers (erroneous results) whenever $S + 2A \leq d - 1$. Therefore, for any encoding scheme that is A secure, it has a Hamming distance of at least $2A + 1$. Consequently, it can tolerate up $2A$ stragglers. Now recall from Lemma 3.6 that to recover a multilinear function f of degree $\deg f$, the maximum number of stragglers any linear encoding scheme can tolerate is upper bounded by $N - (K-1)\deg f - 1$ when $N \geq K \deg f - 1$, and upper bounded by $\lfloor N/K \rfloor - 1$ when $N < K \deg f - 1$. Hence, a computation scheme exists only if $A \leq (N - (K-1)\deg f - 1)/2$ when $N \geq K \deg f - 1$, and $A \leq N/2K - 1/2$ when $N < K \deg f - 1$.

To guarantee T-privacy, the LCC scheme pads the dataset X with additional T random entries before coding; and this amount of randomness is shown to be minimal.

Theorem 4.3 (Optimal randomness)**.** Any computing scheme that universally achieves the (T, A) tradeoff in (4.4)[3] for all linear functions f must use an amount of randomness no less than that of LCC.

Proof of Theorem 4.3

To prove Theorem 4.3, we demonstrate that LCC uses the minimum possible randomness among all linear encoding schemes that achieve the security-privacy tradeoff stated in Theorem 4.1 for linear f. Since the identity map is included in the class of linear functions, one can employ previous results regarding *private storage* to establish a lower bound on the required amount of randomness.

The proof is based on the result in Huang, 2017, Chapter 3. In what follows, an $(n, k, r, z)_{\mathbb{F}_q^t}$ *secure RAID scheme* is a storage scheme over $\mathbb{F}_q^t$ (where $\mathbb{F}_q$ is a field with q elements) in which k message symbols are coded into n storage servers, such that the k message symbols are reconstructible from any $n-r$ servers, and any z servers are information theoretically oblivious to the message symbols. Further, such a scheme is assumed to use v random entries as keys, and by Huang, 2017, Proposition 3.1.1, must satisfy $n - r \geq k + z$.

Theorem 4.4. Huang, 2017, Theorem 3.2.1 A linear rate-optimal $(n, k, r, z)_{\mathbb{F}_q^t}$ secure RAID scheme uses at least zt keys over $\mathbb{F}_q$ (i.e., $v \geq z$).

Clearly, in our scenario $\mathbb{V}$ can be seen as $\mathbb{F}_q^{\dim \mathbb{V}}$ for some q. Further, by setting $N = n$, $T = z$, and $t = \dim \mathbb{V}$, it follows from Theorem 4.4 that any encoding scheme which guarantees information theoretic privacy against sets of T colluding workers must use at least T random entries $\{Z_i\}_{i \in [T]}$.

[3]That is, when two sides of (4.4) are equal.

4.1.3 Comparison with prior works on multiparty computing

Providing security and privacy for multiparty computing (MPC) and machine learning systems is an extensively studied topic. To illustrate the significant role of LCC in secure and private computing, let us consider the celebrated BGW MPC scheme Ben-Or *et al.*, 1988. [4]

Given inputs $\{X_i\}_{i=1}^K$, BGW first uses Shamir's scheme Shamir, 1979a to encode the dataset in a privacy-preserving manner as $P_i(z) = X_i + Z_{i,1}z + \ldots + Z_{i,T}z^T$ for every $i \in [K]$, where $Z_{i,j}$'s are i.i.d. uniformly random variables and T is the number of colluding workers that should be tolerated. The key distinction between the data encoding of BGW scheme and LCC is that we instead use Lagrange polynomials to encode the data. This results in significant reduction in the amount of randomness needed in data encoding (BGW needs KT $z_{i,j}$'s while as we describe in the next section, LCC only needs T amount of randomness).

The BGW scheme will then store $\{P_i(\alpha_\ell)\}_{i=1}^K$ to worker ℓ for every $\ell \in [N]$, given some distinct values $\alpha_1, \ldots, \alpha_N$. The computation is then carried out by evaluating f over *all* stored coded data at the nodes. In the LCC scheme, on the other hand, each worker ℓ only needs to store *one* encoded data $\tilde{X}_\ell$ and compute $f(\tilde{X}_\ell)$. This gives rise to the second key advantage of LCC, which is a factor of K in storage overhead and computation complexity at each worker.

After computation, each worker ℓ in the BGW scheme has essentially evaluated the polynomials $\{f(P_i(z))\}_{i=1}^K$ at $z = \alpha_\ell$, whose degree is at most $\deg f \cdot T$. Hence, if no adversary appears (i.e, $A = 0$), the master can recover all required results $f(P_i(0))$'s, through polynomial interpolation, as long as $N \geq \deg f \cdot T + 1$ workers participated in the computation. It is also possible to use the conventional multi-round BGW, which only requires $N \geq 2T + 1$ workers to ensure T-privacy. However, multiple rounds of computation and communication ($\Omega(\log(\deg f))$ rounds) are needed, which further increases its communication overhead. Note that under the same condition, LCC scheme requires $N \geq \deg f \cdot (K+T-1)+1$ number of workers, which is larger than that of the BGW scheme.

[4] Conventionally, the BGW scheme operates in a multi-round fashion, requiring significantly more communication overhead than one-shot approaches. For simplicity of comparison, we present a modified one-shot version of BGW.

Hence, in overall comparison with the BGW scheme, LCC results in a factor of K reduction in the amount of randomness, storage overhead, and computation complexity, while requiring more workers to guarantee the same level of privacy. This is summarized in Table 4.1.[5]

Table 4.1: Comparison between BGW based designs and LCC. The computational complexity is normalized by that of evaluating f; randomness, which refers to the number of random entries used in encoding functions, is normalized by the length of X_i.

	BGW	LCC
Complexity/worker	K	1
Frac. data/worker	1	$1/K$
Randomness	KT	T
Min. num. of workers	$2T+1$	$\deg f \cdot (K+T-1)+1$

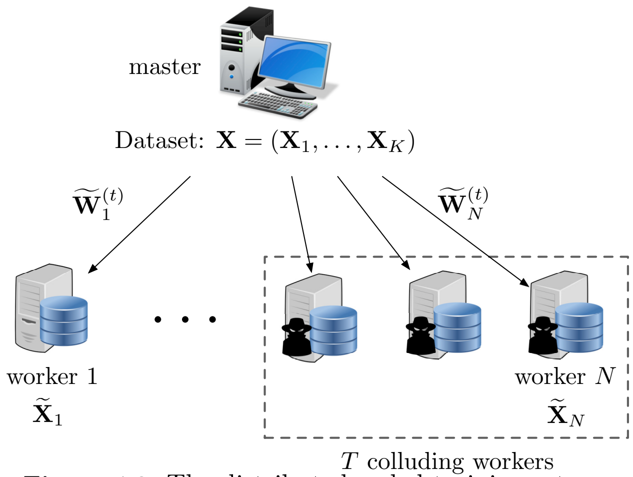
4.2 Privacy preserving machine learning

We now illustrate an application of coded computing, in particular LCC, for privacy-preserving machine learning. We consider a scenario in which a data-owner (e.g., a hospital) wishes to train a logistic regression model by offloading the large volume of data (e.g., healthcare records) and computationally-intensive training tasks (e.g., gradient computations) to N machines over a cloud platform, while ensuring that any collusions between T out of N workers do not leak information about the dataset.

We illustrate a recently proposed scheme, named CodedPrivateML So *et al.*, 2019, that leverages coded computing for this problem. CodedPrivateML has three salient features:

1. it provides strong information-theoretic privacy guarantees for both the training dataset and model parameters.
2. it enables fast training by distributing the training computation load effectively across several workers.
3. it secret shares the dataset and model parameters using coding and information theory principles, which significantly reduces the

[5] A BGW scheme was also proposed in Ben-Or *et al.*, 1988 for secure MPC, however for a substantially different setting. Similarly, a comparison can be made by adapting it to our setting, leading to similar results, which we omit for brevity.


master
Dataset: $\mathbf{X} = (\mathbf{X}_1, \ldots, \mathbf{X}_K)$
$\widetilde{\mathbf{W}}_1^{(t)}$
$\widetilde{\mathbf{W}}_N^{(t)}$
worker 1
$\widetilde{\mathbf{X}}_1$
worker N
$\widetilde{\mathbf{X}}_N$
T colluding workers
Figure 4.2: The distributed coded training setup.

training time.

4.2.1 The CodedPrivateML Framework

We consider the training of a logistic regression model[6]. The dataset is given by a matrix $\mathbf{X} = [\mathbf{x}_1^T \cdots \mathbf{x}_m^T]^T \in \mathbb{R}^{m \times d}$ of m data points with d features and a label vector $\mathbf{y} \in \{0,1\}^m$. Model parameters (weights) $\mathbf{w} \in \mathbb{R}^d$ are obtained by minimizing the cross entropy function,

$$C(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}(-y_i \log \hat{y}_i - (1-y_i)\log(1-\hat{y}_i)) \tag{4.5}$$

where $\hat{y}_i = g(\mathbf{x}_i \cdot \mathbf{w}) \in (0,1)$ is the estimated probability of label i being equal to 1 and $g(z) = 1/(1+e^{-z})$ is the sigmoid function. Problem (4.5) can be solved via gradient descent, through an iterative process that updates the weights in the opposite direction of the gradient $\nabla C(\mathbf{w}) = \frac{1}{m}\mathbf{X}^\top(g(\mathbf{X} \times \mathbf{w}) - \mathbf{y})$. The update function is given by,

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \frac{\eta}{m}\mathbf{X}^\top(g(\mathbf{X} \times \mathbf{w}^{(t)}) - \mathbf{y}) \tag{4.6}$$

where $\mathbf{w}^{(t)}$ holds the estimated parameters from iteration t, η is the learning rate, and $g(\cdot)$ operates element-wise.

We consider a master-worker distributed compute architecture shown in Figure 4.2, where the master offloads the gradient computations in

[6]Analysis applies to linear regression with minor modifications.

(4.6) to N workers. In doing so, the master also wants to protect the privacy of the dataset against any potential collusions between up to T workers, where T is the *privacy parameter* of the system. Initially, the dataset is partitioned into K submatrices $\mathbf{X} = [\mathbf{X}_1^\top \cdots \mathbf{X}_K^\top]^\top$. Parameter $K \in \mathbb{N}$ reflects the amount of parallelization (computation load at each worker is proportional to $1/K$-th of the dataset). The master then creates N *encoded* matrices, $\{\widetilde{\mathbf{X}}_i\}_{i\in[N]}$, by combining the K parts of the dataset with some random matrices to preserve privacy, and sends $\widetilde{\mathbf{X}}_i$ to worker i. At iteration t, master also creates an encoded matrix $\widetilde{\mathbf{W}}_i^{(t)}$ to secret share the current estimate of the weights $\mathbf{w}^{(t)}$ with worker $i \in [N]$, as they can also leak substantial information about the dataset Melis *et al.*, 2019.

The coding strategy should ensure that any subset of T workers can not learn any information, in the information-theoretic sense, about the dataset. Formally, for every subset of workers $\mathcal{T} \subseteq [N]$ with $|\mathcal{T}| \leq T$, we need $I\big(\mathbf{X}; \widetilde{\mathbf{X}}_{\mathcal{T}}, \{\widetilde{\mathbf{W}}_{\mathcal{T}}^{(t)}\}_{t\in[J]}\big) = 0$ where I is the mutual information, J is the number of iterations, and $\widetilde{\mathbf{X}}_{\mathcal{T}}, \{\widetilde{\mathbf{W}}_{\mathcal{T}}^{(t)}\}_{t\in[J]}$ is the collection of coded matrices stored at workers in $\mathcal{T}$.

At each iteration, worker $i \in [N]$ performs its computation locally using $\widetilde{\mathbf{X}}_i$ and $\widetilde{\mathbf{W}}_i^{(t)}$ and sends the result back to the master. After receiving the results from a sufficient number of workers, the master recovers $\mathbf{X}^\top g(\mathbf{X} \times \mathbf{w}^{(t)}) = \sum_{k=1}^{K} \mathbf{X}_k^\top g(\mathbf{X}_k \times \mathbf{w}^{(t)})$ and updates the weights using (4.6).

CodedPrivateML consists of the following four main phases.

Phase 1: Quantization. In order to guarantee information-theoretic privacy, one has to mask the dataset and weights in a finite field $\mathbb{F}$ using uniformly random matrices, so that the added randomness can make each data point appear equally likely. In contrast, the dataset and weights for the training task are defined in the domain of real numbers. Our solution to handle the conversion between the real and finite domains is through the use of stochastic quantization. Accordingly, in the first phase of our system, master quantizes the dataset and weights from the real domain to the domain of integers, and then embeds them in a field $\mathbb{F}_p$ of integers modulo a prime p. The quantized version of the dataset $\mathbf{X}$ is given by $\overline{\mathbf{X}}$. The quantization of the weight vector $\mathbf{w}^{(t)}$, on the other hand, is represented by a matrix $\overline{\mathbf{W}}^{(t)}$, where each column holds an independent stochastic quantization of $\mathbf{w}^{(t)}$. This structure will be important for the convergence of the model. Parameter p is selected to be sufficiently large to avoid wrap-around in computations. Its value depends on the bitwidth of the machine as well as the number of additive and multiplicative operations. For example, in a 64-bit implementation, we select $p = 33554393$ (the largest prime with 25 bits) as explained in our experiments.

Phase 2: Encoding and Secret Sharing. In the second phase, the master partitions the quantized dataset $\overline{\mathbf{X}}$ into K submatrices and encodes them using the LCC approach that we discussed in the previous section. It then sends to worker $i \in [N]$ a coded submatrix $\widetilde{\mathbf{X}}_i \in \mathbb{F}_p^{\frac{m}{K} \times d}$. As we disccused before, this encoding ensures that the coded matrices do not leak any information about the true dataset, even if T workers collude. In addition, the master has to ensure the weight estimations sent to the workers at each iteration do not leak information about the dataset. This is because the weights updated via (4.6) carry information about the whole training set, and sending them directly to the workers may breach privacy. In order to prevent this, at iteration t, master also quantizes the current weight vector $\mathbf{w}^{(t)}$ to the finite field and encodes it again using Lagrange coding.

Phase 3: Polynomial Approximation and Local Computation. In the third phase, each worker performs the computations using its local storage and sends the result back to the master. We note that the workers perform the computations over the encoded data as if they were computing over the true dataset. That is, the structure of the computations are the same for computing over the true dataset versus computing over the encoded dataset. A major challenge is that LCC is designed for distributed polynomial computations. However, the computations in the training phase are not polynomials due to the sigmoid function. We overcome this by approximating the sigmoid with a polynomial of a selected degree r. This allows us to represent the gradient computations in terms of polynomials that can be computed locally by each worker.

Phase 4: Decoding and Model Update. The master collects the results from a subset of fastest workers and decodes the gradient. Then, the master converts the gradient from finite to real domain, updates the weight vector, and secret shares it with the workers for the next round.

Based on this design, we can obtain the following theoretical guarantees for the convergence and privacy of CodedPrivateML. We refer to So *et al.*, 2019 for the details.

Lemma 4.5. Let $\mathbf{p}^{(t)} \triangleq \frac{1}{m}\overline{\mathbf{X}}^{\top}(\bar{g}(\overline{\mathbf{X}}, \overline{\mathbf{W}}^{(t)}) - \mathbf{y})$ be the gradient computation using quantized weights $\overline{\mathbf{W}}^{(t)}$ and degree-r polynomial approximation in CodedPrivateML. Then,

- (Unbiasedness) Vector $\mathbf{p}^{(t)}$ is an asymptotically unbiased estimator of the true gradient. $\mathbb{E}[\mathbf{p}^{(t)}] = \nabla C(\mathbf{w}^{(t)}) + \epsilon(r)$, and $\epsilon(r) \to \mathbf{0}$ as $r \to \infty$ where expectation is taken over the quantization errors,
- (Variance bound) $\mathbb{E}[\|\mathbf{p}^{(t)} - \mathbb{E}[\mathbf{p}^{(t)}]\|_2^2] \leq \frac{1}{2^{-2l_w} m^2} \| \overline{\mathbf{X}} \|_F^2 \triangleq \sigma^2$ where $\| \cdot \|_2$ and $\| \cdot \|_F$ denote the l_2-norm and Frobenius norm, respectively.

Theorem 4.6. Consider the training of a logistic regression model in a distributed system with N workers with dataset $\mathbf{X} = (\mathbf{X}_1, \dots, \mathbf{X}_K)$, initial weights $\mathbf{w}^{(0)}$, and constant step size $\eta = 1/L$ where $L \triangleq \frac{1}{4}\|\overline{\mathbf{X}}\|_2^2$. For any $N \geq (2r+1)(K+T-1)+1$, CodedPrivateML guarantees,

- (Convergence) $\mathbb{E}[C(\frac{1}{J}\sum_{t=0}^{J} \mathbf{w}^{(t)})] - C(\mathbf{w}^*) \leq \frac{\| \mathbf{w}^{(0)} - \mathbf{w}^* \|^2}{2\eta J} + \eta\sigma^2$ in J iterations, with σ^2 from Lemma 4.5,
- (Privacy) $\mathbf{X}$ remains information-theoretically private against any T colluding workers, i.e., $I\Big(\mathbf{X}; \widetilde{\mathbf{X}}_{\mathcal{T}}, \{\widetilde{\mathbf{W}}_{\mathcal{T}}^{(t)}\}_{t\in[J]}\Big) = 0$, $\forall \mathcal{T} \subset [N]$, $|\mathcal{T}| \leq T$,

Theorem 4.6 reveals an important trade-off between privacy (T) and parallelization (K), that is, each additional worker can be utilized either for more privacy or a faster training.

4.2.2 Experimental Evaluation of CodedPrivateML

The performance of CodedPrivateML had been experimentally demonstrated in So *et al.*, 2019 over Amazon EC2 Cloud Platform for training a logistic regression model for image classification. In particular, CodedPrivateML has been used for training the logistic regression model from (4.5) for binary image classification on the CIFAR-10 Krizhevsky and Hinton, 2009 and GISETTE Guyon *et al.*, 2005 datasets to experimentally examine two things: the accuracy of CodedPrivateML and the performance gain in terms of training time over two MPC-based benchmarks. The first one is based on the well-known BGW protocol Ben-Or *et al.*, 1988, whereas the second one is a more recent protocol from Beerliova-Trubiniova and Hirt, 2008; Damgård and Nielsen, 2007 that trade-offs offline calculations for a more efficient implementation. Both baselines utilize Shamir's secret sharing scheme Shamir, 1979b where the dataset is secret shared among the N workers.

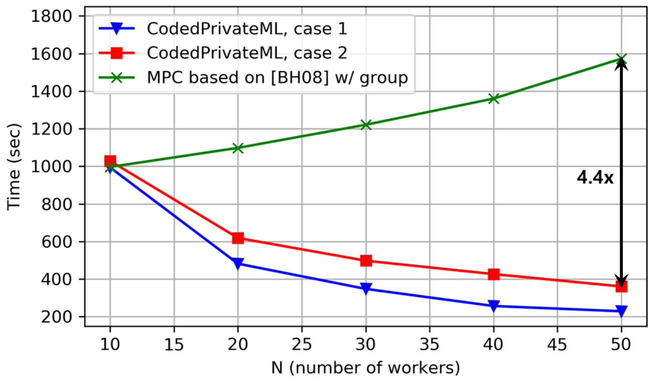
CodedPrivateML parameters. There are several system parameters in CodedPrivateML that should be set. Given that a 64-bit implementation was used in So *et al.*, 2019, the field size was selected to be $p = 33554393$, which is the largest prime with 25 bits to avoid an overflow on intermediate multiplications.

One needs to also set the parameter r, the degree of the polynomial for approximating the sigmoid function. Both $r = 1$ and $r = 2$ was soncisdered in So *et al.*, 2019, and it was empirically observed that the degree one approximation achieves good accuracy. Finally, one needs to select T (privacy threshold) and K (amount of parallelization) in CodedPrivateML. As stated in Theorem 4.6, these parameters should satisfy $N \geq (2r + 1)(K + T - 1) + 1$. Given the choice of $r = 1$, two cases can be considered:

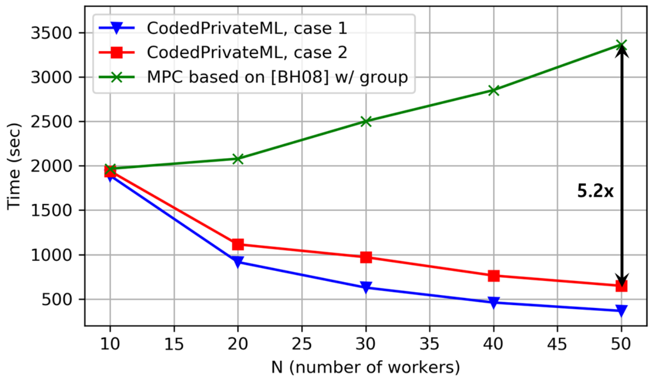
- **Case 1 (maximum parallelization).** All resources allocated for parallelization (faster training) by setting $K = \lfloor \frac{N-1}{3} \rfloor$, $T = 1$,

- **Case 2 (equal parallelization & privacy).** Resources split almost equally between parallelization & privacy, i.e., $T = \lfloor \frac{N-3}{6} \rfloor, K = \lfloor \frac{N+2}{3} \rfloor - T$.

With these parameters, the training time of CodedPrivateML has been measured while increasing the number of workers N gradually. The results are demonstrated in Figure 4.3, which shows the comparison of CodedPrivateML with the [BH08] protocol from Beerliova-Trubiniova and Hirt, 2008, which was the faster of the two benchmarks. In particular,

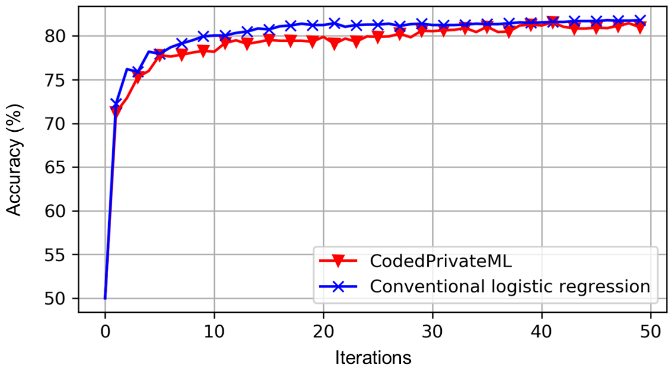


(a) CIFAR-10 (for accuracy 81.35% with 50 iterations)

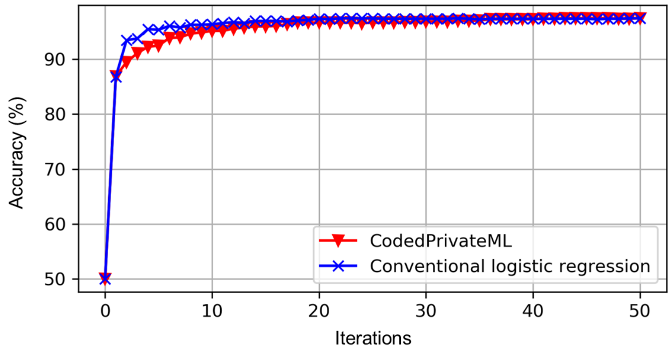


(b) GISETTE (for accuracy 97.50% with 50 iterations)

Figure 4.3: Performance gain of CodedPrivateML over the MPC baseline ([BH08] from Beerliova-Trubiniova and Hirt, 2008). The plot shows the total training time for different number of workers N.



(a) CIFAR-10 dataset, binary classification between *car* and *plain* images (using 9019 samples for training and 2000 samples for testing).



(b) GISETTE dataset, binary classification between digits 4 and 9 images (using 6000 samples for training and 1000 samples for testing).

Figure 4.4: Comparison of the accuracy of CodedPrivateML (demonstrated for Case 2 and $N = 50$ workers) vs conventional logistic regression that uses the sigmoid function without quantization.

Table 4.2: (CIFAR-10) Breakdown of total runtime for $N = 50$.

Protocol	Enc. time	Comm. time	Comp. time	Total
MPC using [BGW88]	202.78s	31.02s	7892.42s	8127.07s
MPC using [BH08]	201.08s	30.25s	1326.03s	1572.34s
CodedPrivateML (Case 1)	59.93s	4.76s	141.72s	229.07s
CodedPrivateML (Case 2)	91.53s	8.30s	235.18s	361.08s

we make the following observations. [7]

- CodedPrivateML provides substantial speedup over the MPC baselines, in particular, up to $4.4\times$ and $5.2\times$ with the CIFAR-10 and GISETTE datasets, respectively, while providing the same privacy threshold as the benchmarks ($T = \lfloor \frac{N-3}{6} \rfloor$ for Case 2). Table 4.2 demonstrates the breakdown of the total runtime with the CIFAR-10 dataset for $N = 50$ workers. In this scenario, CodedPrivateML provides significant improvement in all three categories of dataset encoding and secret sharing; communication time between the workers and the master; and the computation time. Main reason for this is that, in the MPC baselines, the size of the data processed at each worker is one third of the original dataset, while in CodedPrivateML it is $1/K$-th of the dataset. This reduces the computational overhead of each worker while computing matrix multiplications as well as the communication overhead between the master and workers. We also observe that a higher amount of speedup is achieved as the dimension of the dataset becomes larger (CIFAR-10 vs. GISETTE datasets), suggesting CodedPrivateML to be well-suited for data-intensive training tasks where parallelization is essential.

- The total runtime of CodedPrivateML decreases as the number of workers increases. This is again due to the parallelization gain of CodedPrivateML (i.e., increasing K while N increases). This is not achievable in conventional MPC baselines, since the size of data processed at each worker is constant for all N.

[7]For $N = 10$, all schemes have similar performance because the total amount of data stored at each worker is one third of the size of whole dataset ($K = 3$ for CodedPrivateML and $G = 3$ for the benchmark).

- CodedPrivateML provides up to 22.5× speedup over the BGW protocol Ben-Or *et al.*, 1988, as shown in Table 4.2 for the CIFAR-10 dataset with $N = 50$ workers. This is due to the fact that BGW requires additional communication between the workers to execute a degree reduction phase for every multiplication operation.

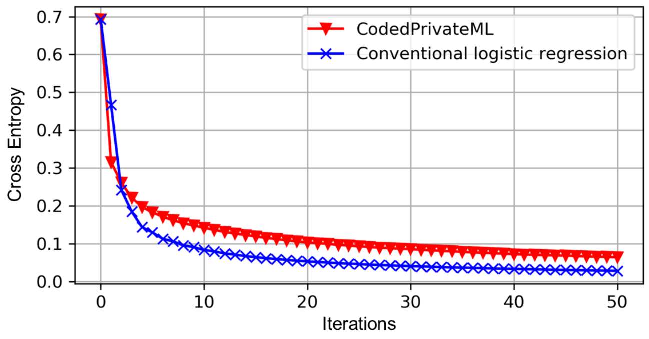


Figure 4.5: Convergence of CodedPrivateML (demonstrated for Case 2 and $N = 50$ workers) vs conventional logistic regression (using the sigmoid function without polynomial approximation or quantization).

The accuracy and convergence of CodedPrivateML was also experimentally analyzed in So *et al.*, 2019. Figure 4.4(a) illustrates the test accuracy of the binary classification problem between *plane* and *car* images for the CIFAR-10 dataset. With 50 iterations, the accuracy of CodedPrivateML with degree one polynomial approximation and conventional logistic regression are 81.35% and 81.75%, respectively. Figure 4.4(b) shows the test accuracy for binary classification between digits 4 and 9 for the GISETTE dataset. With 50 iterations, the accuracy of CodedPrivateML with degree one polynomial approximation and conventional logistic regression has the same value of 97.5%. Hence, CodedPrivateML has comparable accuracy to conventional logistic regression while being privacy preserving.

Figure 4.5 presents the cross entropy loss for CodedPrivateML versus the conventional logistic regression model for the GISETTE dataset. The latter setup uses the sigmoid function and no polynomial approximation, in addition, no quantization is applied to the dataset or the weight vectors. We observe that CodedPrivateML achieves convergence with comparable rate to conventional logistic regression, while being privacy preserving.

4.3 Related works and open problems

The security and privacy issue has been extensively studied in the literature of secure multiparty computing and distributed machine learning/data mining Ben-Or *et al.*, 1988; Cramer *et al.*, 2001; Lindell, 2005; Cramer *et al.*, 2015; Mohassel and Zhang, 2017. For instance, the celebrated BGW scheme Ben-Or *et al.*, 1988 employs Shamir's scheme Shamir, 1979a to privately share intermediate results between parties. As we have elaborated in Section 4.1, the proposed LCC scheme significantly improves the BGW in the required storage overhead, computational complexity, and the amount of injected randomness (Table 4.1).

There have also been several other recent works have on coded computing under privacy and security constraints. Extending the research works on secure *storage* (see, e.g., Pawar *et al.*, 2011; Shah *et al.*, 2011; Rawat *et al.*, 2014), staircase codes Bitar *et al.*, 2018 have been proposed to combat stragglers in linear computations (e.g., matrix-vector multiplications) while preserving data privacy, which was shown to reduce the computation latency of the schemes based on classical secret sharing strategies Shamir, 1979a; McEliece and Sarwate, 1981. The proposed Lagrange Coded Computing scheme in this chapter generalizes the staircase codes beyond linear computations. Even for the linear case, LCC guarantees data privacy against T colluding workers by introducing less randomness than Bitar *et al.*, 2018 (T rather than $TK/(K-T)$).

Beyond linear computations, a secure coded computing scheme was proposed in Yang and Lee, 2019 to achieve data security for distributed matrix-matrix multiplication. Leveraging the polynomial code proposed in Yu *et al.*, 2017b, the secure computing scheme in Yang and Lee, 2019 achieves the order-optimal recovery threshold, while preserving data privacy at each worker (i.e., $T = 1$). For computing more general class of matrix polynomials, Nodehi and Maddah-Ali, 2018 has combined ideas from the BGW scheme and Yu *et al.*, 2017b to form the so-called *polynomial sharing*, a private coded computation scheme for arbitrary matrix polynomials. However, polynomial sharing inherits the undesired BGW property of performing a communication round for *every* linear and *every* bilinear operation in the polynomial; a feature that drastically increases the communication overhead, and is circumvented by the one-shot approach of LCC.

DRACO Chen *et al.*, 2018 was proposed as a secure distributed training algorithm that is robust to Byzantine faults. Since DRACO is designed for general gradient computations, it employs a blackbox approach, i.e., the coding is applied on the gradients computed from uncoded data, but not on the data itself, which is similar to the gradient coding techniques Tandon *et al.*, 2017; Halbawi *et al.*, 2017; Raviv *et al.*, 2017; Ye and Abbe, 2018; Li *et al.*, 2018c designed primarily for stragglers. Hence, the inherent algebraic structure of the gradients is ignored. For this approach, Chen *et al.*, 2018 show that a $2A+1$ *multiplicative* factor of redundant computations is necessary to be robust to A Byzantine workers. For the proposed LCC however, the blackbox approach is disregarded in favor of an algebraic one, and consequently, a $2A$ *additive* factor suffices.

We end this section by highlighting a few other interesting directions for future research.

Extension to real-field computations. Most of the works that we have described so far rely on quantizing the data into a finite field, so that the coded computing approaches for secure and private computing can then be employed. These approaches, however, can result in substantial accuracy losses due to quantization, fixed-point representation of the data, and computation overflows (see e.g., Fahim and Cadambe, 2019). An important research direction would be to develop coded computing techniques for secure and private computing in the real-field domain.

Application to deep neural network training. Another barrier in the theory of secure and private computing and machine learning is their efficient generalization to non-polynomial computations. Many non-linear threshold functions that arise in machine learning, in particular rectified linear unit (ReLU) functions in deep neural networks, cannot be approximated well with low degree polynomials. Therefore, finding new coded computing approaches that enable secure and private computing for such classes of computations would be of great interest.

Application to large-scale federated learning. Another interesting application of coding for secure and private computing would be the federated learning problem that has attracted a lot of attention recently. Federated learning is an emerging approach that enables model training over a large volume of decentralized data residing in mobile devices, while protecting the privacy of the individual users McMahan *et al.*, 2017; Bonawitz *et al.*, 2016; Bonawitz *et al.*, 2017; Kairouz *et al.*, 2019. A major bottleneck in scaling secure federated learning to a large number of users is the overhead of secure model aggregation across many users. An interesting direction is understand the role of coded computing for efficient, scalable, and secure model aggregation in large federated systems. Some promising work in this direction is initiated in So *et al.*, 2020.

Application to blockchain systems. Secure, private, and verifiable computing are also of great importance in decentralized blockchain systems. Today's blockchain designs suffer from a trilemma claiming that no blockchain system can simultaneously achieve decentralization, security, and performance scalability. For current blockchain systems, as more nodes join the network, the efficiency of the system (computation, communication, and storage) stays constant at best. A leading idea for enabling blockchains to scale efficiency is the notion of sharding: different subsets of nodes handle different portions of the blockchain, thereby reducing the load for each individual node. However, existing sharding proposals achieve efficiency scaling by compromising on trust - corrupting the nodes in a given shard will lead to the permanent loss of the corresponding portion of data. Coded computing can provide an effective approach for overcoming such barriers in distributed systems. Coded computing can also provide new approaches for dealing with the issues of computation verification and data availability in decentralized blockchain systems. Several recent works in these directions have been initiated in Li *et al.*, 2018e; Yu *et al.*, 2020; Sahraei and Avestimehr, 2019; Kadhe *et al.*, 2019; Sahraei *et al.*, 2019; Mitra and Dolecek, 2019; Choi *et al.*, 2019.

Acknowledgements

We would like to thank several colleagues and students whose comments and collaborations greatly helped the monograph. In particular we would like to acknowledge the collaborations with B. Guler, M. Maddah Ali, P. Mohassel, R. Pedarsani, N. Raviv, J. So, and Q. Yu. The authors would also like to thank the anonymous reviewer for her/his valuable comments.

We would like to also thank several funding agencies for enabling this research. In particular, the content of this monograph is based upon work supported by National Science Foundation (grants CCF-1703575 and CCF-1763673), Defense Advanced Research Projects Agency (DARPA) (award HR001117C0053), Office of Naval Research (award N00014-16-1-2189), Army Research Office (award W911NF1810400), and Intel Labs.

Appendices

A

Proof of Lemma 3.6 Lower bound on the recovery threshold of computing multilinear functions

Before we start the proof, we let $K^*_f(K, N)$ denote the minimum recovery threshold given the function f, the number of computations K, and the number of workers N.

We now proceed to prove Lemma 3.6 by induction.

(a) When $d = 1$, then f is a linear function, and we aim to prove $K^*_f(K, N) \geq K$. Assuming the opposite, we can find a computation design such that for a subset $\mathcal{N}$ of at most $K - 1$ workers, there is a decoding function that computes all $f(X_i)$'s given the results from workers in $\mathcal{N}$.

Because the encoding functions are linear, we can thus find a non-zero vector $(a_1, ..., a_K) \in \mathbb{F}^K$ such that when $X_i = a_i V$ for any $V \in \mathbb{V}$, the coded variable $\tilde{X}_i$ stored by any worker in $\mathcal{N}$ equals 0. This leads to a fixed output from the decoder. On the other hand, because f is assumed to be non-zero, the computing results $\{f(X_i)\}_{i\in[K]}$ is variable for different values of V, which leads to a contradiction. Hence, we have $K^*_f(K, N) \geq K$.

(b) Suppose we have a matching converse for any multilinear function with $d = d_0$. We now prove the lower bound for any non-zero multilinear function f of degree $d_0 + 1$. The proof idea is to construct a multilinear function f' with degree d_0 based on function f, and to lower bound the minimum recovery threshold of f using that of f'. More specifically, this is done by showing that given any computation design for function f, a computation design can also be developed for the corresponding f', which achieves a recovery threshold that is related to that of the scheme for f.

In particular, for any non-zero function $f(X_{i,1}, X_{i,2}, ..., X_{i,d_0+1})$, we can find $V \in \mathbb{V}$, such that $f(X_{i,1}, X_{i,2}, ..., X_{i,d_0}, V)$ as a function of $(X_{i,1}, X_{i,2}, ..., X_{i,d_0})$ is non-zero. We define $f'(X_{i,1}, X_{i,2}, ..., X_{i,d_0}) = f(X_{i,1}, X_{i,2}, ..., X_{i,d_0}, V)$, which is a multilinear function with degree d_0. Given parameters K and N, we now develop a computation strategy for f' for a dataset of K inputs and a cluster of $N' \triangleq N - K$ workers, which achieves a recovery threshold of $K^*_f(K, N) - (K - 1)$. We construct this computation strategy based on an encoding strategy of f that achieves the recovery threshold $K^*_f(K, N)$. Because the encoding functions are linear, we consider the encoding matrix, denoted by $G \in \mathbb{F}^{K \times N}$, and defined as the coefficients of the encoding functions $\tilde{X}_i = \sum_{j=1}^{K} X_j G_{ji}$. Following the same arguments we used in the $d = 1$ case, the left null space of G must be $\{0\}$. Consequently, the rank of G equals K, and we can find a subset $\mathcal{K}$ of K workers such that the corresponding columns of G form a basis of $\mathbb{F}^K$. We construct a computation scheme for f' with $N' \triangleq N - K$ workers, each of whom stores the coded version of $(X_{i,1}, X_{i,2}, \ldots, X_{i,d_0})$ that is stored by a unique respective worker in $[N] \setminus \mathcal{K}$ in the computation scheme of f.

Now it suffices to prove that the above construction achieves a recovery threshold of $K^*_f(K, N) - (K - 1)$. Equivalently, we need to prove that given any subset $\mathcal{S}$ of $[N] \backslash \mathcal{K}$ of size $K^*_f(K, N) - (K - 1)$, the values of $f(X_{i,1}, X_{i,2}, ..., X_{i,d_0}, V)$ for $i \in [K]$ are decodable from the computing results of workers in $\mathcal{S}$.

We now exploit the decodability of the computation design for function f. For any $j \in \mathcal{K}$, the set $\mathcal{S} \cup \mathcal{K} \backslash \{j\}$ has size $K^*_f(K, N)$. Consequently, for any vector $\boldsymbol{a} = (a_1, ..., a_K) \in \mathbb{F}^K$, by letting $X_{i,d_0+1} = a_i V$, we have that $\{a_i f(X_{i,1}, X_{i,2}, ..., X_{i,d_0}, V)\}_{i \in [K]}$ is decodable given the computing results from workers in $\mathcal{S} \cup \mathcal{K} \backslash \{j\}$. Moreover, for any $j \in [K]$, let $\boldsymbol{a}^{(j)} \in \mathbb{F}^K$ be a non-zero vector that is orthogonal to all columns of G with indices in $\mathcal{K}\backslash\{j\}$, workers in $\mathcal{K}\backslash\{j\}$ would store 0 for the X_{i,d_0+1} entry, and return constant 0 due to the multilinearity of f. Consequently, any $\{a_i^{(j)} f(X_{i,1}, X_{i,2}, ..., X_{i,d_0}, V)\}_{i\in[K]}$ is decodable from the computing results from workers in $\mathcal{S}$.

Because columns of G with indices in $\mathcal{K}$ form a basis of $\mathbb{F}^K$, the vectors $\boldsymbol{a}^{(j)}$ for $j \in \mathcal{K}$ also from a basis. Consequently, $f'(X_{i,1}, X_{i,2}, ..., X_{i,d_0})$, which equals $f(X_{i,1}, X_{i,2}, ..., X_{i,d_0}, V)$, is also decodable given results from workers in $\mathcal{S}$ for any $i \in [K]$. On the other hand, note that the computing results for each worker in $\mathcal{S}$ given each $\boldsymbol{a}^{(j)}$ can also be computed using the results from the same workers when computing f'. Hence, the decoder for function f' can first recover the computing results for workers in $\mathcal{S}$ for function f, and then proceed to decoding the final result. Thus we have completed the proof of decodability.

To summarize, we have essentially proved that $K_f^*(K, N) - (K-1) \geq K_{f'}^*(K, N-K)$ when $N \geq 2K$, and $K_f^*(K, N) - (K-1) > N - K$ otherwise. Hence, we verify that the converse bound for $K_f^*(K, N)$ holds for any function f with degree $d_0 + 1$ given the above result and the induction assumption.

(c) Thus, the converse bound in Lemma 3.6 holds for any $d \in \mathbb{N}^+$.

References

Agarwal, A. and A. Mazumdar. 2016. “Local Partial Clique and Cycle Covers for Index Coding”. In: *2016 IEEE Globecom Workshops (GC Wkshps)*. 1–6.

Ahlswede, R., N. Cai, S.-Y. R. Li, and R. W. Yeung. 2000. “Network information flow”. *IEEE Transactions on Information Theory*. 46(4): 1204–1216.

Ahmad, F., S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. 2012. “Tarazu: optimizing MapReduce on heterogeneous clusters”. In: *ACM SIGARCH Computer Architecture News*. Vol. 40. No. 1. 61–74.

Aktas, M. F., P. Peng, and E. Soljanin. 2018. “Straggler Mitigation by Delayed Relaunch of Tasks”. *ACM SIGMETRICS Performance Evaluation Review*. 45(2): 224–231.

Alistarh, D., D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. 2017. “QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding”. *Advances in neural information processing systems (NIPS)*: 1707–1718.

Alpatov, P., G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, and Y.-J. J. Wu. 1997. “PLAPACK: parallel linear algebra package design overview”. In: *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*. ACM. 1–16.

"Amazon Elastic Compute Cloud (EC2)". https://aws.amazon.com/ec2/. Accessed on Jan. 30, 2018.

Ananthanarayanan, G., A. Ghodsi, S. Shenker, and I. Stoica. 2013. "Effective straggler mitigation: Attack of the clones". In: *10th USENIX Symposium on Networked Systems Design and Implementation*. 185–198.

Ananthanarayanan, G., S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. 2010. "Reining in the Outliers in Map-Reduce Clusters using Mantri." In: *OSDI*. Vol. 10. No. 1. 24.

"Apache Hadoop". http://hadoop.apache.org. Accessed on Jan. 30, 2018.

Arbabjolfaei, F. and Y. Kim. 2018. *Fundamentals of Index Coding*.

Attia, M. A. and R. Tandon. 2016. "Information Theoretic Limits of Data Shuffling for Distributed Learning". In: *IEEE Global Communications Conference (GLOBECOM)*. 1–6. DOI: 10.1109/GLOCOM.2016.7841903.

Baktir, S. and B. Sunar. 2006. "Achieving efficient polynomial multiplication in fermat fields using the fast fourier transform". In: *Proceedings of the 44th annual Southeast regional conference*. ACM. 549–554.

Ballard, G., E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. 2014. "Communication lower bounds and optimal algorithms for numerical linear algebra". *Acta Numerica*. 23: 1–155.

Bar-Yossef, Z., Y. Birk, T. Jayram, and T. Kol. 2011. "Index coding with side information". *IEEE Transactions on Information Theory*. 57(3): 1479–1494.

Becker, K. and U. Wille. 1998. "Communication complexity of group key distribution". In: *Proceedings of the 5th ACM conference on Computer and communications security*. 1–6.

Beerliova-Trubiniova, Z. and M. Hirt. 2008. "Perfectly-secure MPC with linear communication complexity". In: *Theory of Crypto. Conf.* Springer. 213–230.

Ben-Or, M., S. Goldwasser, and A. Wigderson. 1988. "Completeness theorems for non-cryptographic fault-tolerant distributed computation". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1–10.

Bernstein, J., Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar. 2018. "signSGD: Compressed Optimisation for Non-Convex Problems". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by J. Dy and A. Krause. Vol. 80. *Proceedings of Machine Learning Research*. Stockholmsmässan, Stockholm Sweden: PMLR. 560–569. URL: http://proceedings.mlr.press/v80/bernstein18a.html.

Birk, Y. and T. Kol. 2006. "Coding on demand by an informed source (ISCOD) for efficient broadcast of different supplemental data to caching clients". *IEEE Transactions on Information Theory*. 52(6): 2825–2830.

Bitar, R., P. Parag, and S. E. Rouayheb. 2018. "Minimizing Latency for Secure Coded Computing Using Secret Sharing via Staircase Codes". *e-print arXiv:1802.02640*.

Blackford, L. S., J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, *et al.* 1997. *ScaLAPACK users' guide*. SIAM.

Blanchard, P., E. M. E. Mhamdi, R. Guerraoui, and J. Stainer. 2017a. "Byzantine-tolerant machine learning". *arXiv preprint arXiv:1703.02757*.

Blanchard, P., R. Guerraoui, J. Stainer, *et al.* 2017b. "Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent". In: *Advances in Neural Information Processing Systems*. 118–128.

Bogdanov, D., S. Laur, and J. Willemson. 2008. "Sharemind: A Framework for Fast Privacy-Preserving Computations". In: *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security. ESORICS '08*. M&#225;laga, Spain: Springer-Verlag. 192–206. ISBN: 978-3-540-88312-8. DOI: 10.1007/978-3-540-88313-5_13. URL: http://dx.doi.org/10.1007/978-3-540-88313-5_13.

Bonawitz, K., V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. 2016. "Practical secure aggregation for federated learning on user-held data". *Conference on Neural Information Processing Systems.*

Bonawitz, K., V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. 2017. "Practical secure aggregation for privacy-preserving machine learning". In: *ACM SIGSAC Conf. on Comp. and Comm. Security.* ACM. 1175–1191.

Bonomi, F., R. Milito, J. Zhu, and S. Addepalli. 2012. "Fog computing and its role in the internet of things". In: *Proceedings of the 1st edition of the MCC workshop on Mobile cloud computing.* ACM. 13–16.

Charles, Z., D. Papailiopoulos, and J. Ellenberg. 2017. "Approximate gradient coding via sparse random graphs". *arXiv preprint arXiv:1711.06771.*

Chaubey, M. and E. Saule. 2015. "Replicated data placement for uncertain scheduling". In: *IEEE International Parallel and Distributed Processing Symposium Workshop.* 464–472.

Chen, L., Z. Charles, D. Papailiopoulos, *et al.* 2018. "DRACO: Robust Distributed Training via Redundant Gradients". *e-print arXiv:1803.09877.*

Chiang, M. and T. Zhang. 2016. "Fog and IoT: An Overview of Research Opportunities". *IEEE Internet of Things Journal.*

Chilimbi, T. M., Y. Suzue, J. Apacible, and K. Kalyanaraman. 2014. "Project Adam: Building an Efficient and Scalable Deep Learning Training System." In: *11th USENIX Symposium on Operating Systems Design and Implementation.* Vol. 14. 571–582.

Choi, B., J.-y. Sohn, D.-J. Han, and J. Moon. 2019. "Scalable network-coded PBFT consensus algorithm". In: *2019 IEEE International Symposium on Information Theory (ISIT).* IEEE. 857–861.

Choi, J., J. Dongarra, and D. Walker. 1996. "PB-BLAS: A set of parallel block basic linear algebra subprograms". *Concurrency: Practice and Experience.* 8(7): 517–535. ISSN: 1040-3108.

Chowdhury, M., M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. 2011. "Managing data transfers in computer clusters with orchestra". *ACM SIGCOMM Computer Communication Review*. 41(4): 98–109.

Cramer, R., I. Damgård, and J. B. Nielsen. 2001. "Multiparty computation from threshold homomorphic encryption". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 280–300.

Cramer, R., I. B. Damgrd, and J. B. Nielsen. 2015. *Secure Multiparty Computation and Secret Sharing*. 1st. New York, NY, USA: Cambridge University Press. ISBN: 1107043050, 9781107043053.

Dalcin, L. D., R. R. Paz, P. A. Kler, and A. Cosimo. 2011. "Parallel distributed computing using python". *Advances in Water Resources*. 34(9): 1124–1139.

Damgård, I. and J. B. Nielsen. 2007. "Scalable and unconditionally secure multiparty computation". In: *International Cryptology Conf.* Springer. 572–590.

Dean, J. and L. A. Barroso. 2013. "The tail at scale". *Communications of the ACM*. 56(2): 74–80.

Dean, J. and S. Ghemawat. 2004. "MapReduce: Simplified data processing on large clusters". *Sixth USENIX Symposium on Operating System Design and Implementation*. Dec.

Demmel, J., L. Grigori, M. Hoemmen, and J. Langou. 2012. "Communication-optimal parallel and sequential QR and LU factorizations". *SIAM Journal on Scientific Computing*. 34(1): A206–A239.

Dimakis, A. G., J. Wang, and K. Ramchandran. 2007. "Unequal growth codes: Intermediate performance and unequal error protection for video streaming". In: *Multimedia Signal Processing, 2007. MMSP 2007. IEEE 9th Workshop on*. IEEE. 107–110.

"Distributed Algorithms and Optimization Lecture Notes". https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture16/Pregel_GraphX.pdf. Accessed: 2018-07-11.

Dutta, S., Z. Bai, T. M. Low, and P. Grover. 2019. "CodeNet: Training large scale neural networks in presence of soft-errors". *arXiv preprint arXiv:1903.01042*.

Dutta, S., V. Cadambe, and P. Grover. 2016. “Short-dot: Computing large linear transforms distributedly using coded short dot products”. *NIPS*: 2100–2108.

Dutta, S., V. Cadambe, and P. Grover. 2017. “Coded convolution for parallel and distributed computing within a deadline”. In: *IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2403–2407.

Ekanayake, J., H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. 2010. “Twister: a runtime for iterative MapReduce”. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. June: 810–818.

Ezzeldin, Y. H., M. Karmoose, and C. Fragouli. 2017. “Communication vs distributed computation: an alternative trade-off curve”. In: *IEEE Information Theory Workshop (ITW)*. IEEE. 279–283.

Fahim, M. and V. R. Cadambe. 2019. “Numerically Stable Polynomially Coded Computing”. In: *2019 IEEE International Symposium on Information Theory (ISIT)*. 3017–3021.

Fahim, M., H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover. 2017. “On the optimal recovery threshold of coded matrix multiplication”. In: *55th Annual Allerton Conference*. IEEE. 1264–1270.

Al-Fares, M., S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. 2010. “Hedera: Dynamic Flow Scheduling for Data Center Networks”. *7th USENIX Symposium on Networked Systems Design and Implementation*. Apr.

Ferdinand, N. and S. C. Draper. 2018. “Hierarchical Coded Computation”. In: *2018 IEEE International Symposium on Information Theory (ISIT)*. 1620–1624.

Gardner, K., S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia. 2015. “Reducing latency via redundant requests: Exact analysis”. *ACM SIGMETRICS Performance Evaluation Review*. 43(1): 347–360.

Gemulla, R., E. Nijkamp, P. J. Haas, and Y. Sismanis. 2011. "Large-scale matrix factorization with distributed stochastic gradient descent". In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 69–77.

Greenberg, A., J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. 2009. "VL2: a scalable and flexible data center network". *ACM SIGCOMM computer communication review*. 39(4): 51–62.

Guo, Y., J. Rao, and X. Zhou. 2013. "iShuffle: Improving Hadoop Performance with Shuffle-on-Write". In: *Proceedings of the 10th International Conference on Autonomic Computing*. 107–117.

Gupta, V., S. Wang, T. Courtade, and K. Ramchandran. 2018. "OverSketch: Approximate Matrix Multiplication for the Cloud". In: *2018 IEEE International Conference on Big Data (Big Data)*. 298–304.

Guyon, I., S. Gunn, A. Ben-Hur, and G. Dror. 2005. "Result Analysis of the NIPS 2003 Feature Selection Challenge". In: *Advances in Neural Inf. Processing Systems*. 545–552.

Haddadpour, F. and V. R. Cadambe. 2018. "Codes for Distributed Finite Alphabet Matrix-Vector Multiplication". In: *2018 IEEE International Symposium on Information Theory (ISIT)*. 1625–1629.

"Hadoop TeraSort". https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html. Accessed on Jan. 30, 2018.

Halbawi, W., N. Azizan-Ruhi, F. Salehi, and B. Hassibi. 2017. "Improving Distributed Gradient Descent Using Reed-Solomon Codes". *e-print arXiv:1706.05436*.

Halpern, J. and V. Teague. 2004. "Rational secret sharing and multiparty computation". In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM. 623–632.

He, K., X. Zhang, S. Ren, and J. Sun. 2016. "Deep residual learning for image recognition". *IEEE conference on computer vision and pattern recognition*: 770–778.

Ho, T., R. Koetter, M. Medard, D. R. Karger, and M. Effros. 2003. "The benefits of coding over routing in a randomized setting". *IEEE International Symposium on Information Theory*. June: 442–. DOI: 10.1109/ISIT.2003.1228459.

Huang, K.-H. and J. A. Abraham. 1984. "Algorithm-Based Fault Tolerance for Matrix Operations". *IEEE Transactions on Computers*. C-33(6): 518–528. ISSN: 0018-9340. DOI: 10.1109/TC.1984.1676475.

Huang, L., A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. 2011. "Adversarial machine learning". In: *Proceedings of the 4th ACM workshop on Security and artificial intelligence*. ACM. 43–58.

Huang, W. 2017. "Coding for Security and Reliability in Distributed Systems". *PhD thesis*. California Institute of Technology.

Jahani-Nezhad, T. and M. A. Maddah-Ali. 2019. "CodedSketch: Coded Distributed Computation of Approximated Matrix Multiplication". In: *2019 IEEE International Symposium on Information Theory (ISIT)*. 2489–2493.

Jeong, H., T. M. Low, and P. Grover. 2018. "Masterless Coded Computing: A Fully-Distributed Coded FFT Algorithm". In: *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 887–894.

Ji, M., G. Caire, and A. F. Molisch. 2016. "Fundamental limits of caching in wireless D2D networks". *IEEE Transactions on Information Theory*. 62(2): 849–869. ISSN: 0018-9448. DOI: 10.1109/TIT.2015.2504556.

Joshi, G., E. Soljanin, and G. Wornell. 2017. "Efficient redundancy techniques for latency reduction in cloud systems". *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*. 2(2): 12.

Jou, J.-Y. and J. A. Abraham. 1986. "Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures". *Proceedings of the IEEE*. 74(5): 732–741. ISSN: 0018-9219. DOI: 10.1109/PROC.1986.13535.

Kadhe, S., J. Chung, and K. Ramchandran. 2019. "SeF: A secure fountain architecture for slashing storage costs in blockchains". *arXiv preprint arXiv:1906.12140*.

Kairouz, P., H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, *et al.* 2019. “Advances and open problems in federated learning”. *arXiv preprint arXiv:1912.04977*.

Kamra, A., V. Misra, J. Feldman, and D. Rubenstein. 2006. “Growth codes: Maximizing sensor network data persistence”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 36. No. 4. ACM. 255–266.

Karakus, C., Y. Sun, S. Diggavi, and W. Yin. 2017. “Straggler mitigation in distributed optimization through data encoding”. In: *Advances in Neural Information Processing Systems*. 5440–5448.

Karamchandani, N., U. Niesen, M. A. Maddah-Ali, and S. Diggavi. 2014. “Hierarchical coded caching”. *IEEE International Symposium on Information Theory*. June: 2142–2146.

Kedlaya, K. S. and C. Umans. 2011. “Fast polynomial factorization and modular composition”. *SIAM Journal on Computing*. 40(6): 1767–1802.

Kiamari, M., C. Wang, and A. S. Avestimehr. 2017. “On Heterogeneous Coded Distributed Computing”. *IEEE GLOBECOM*. Dec.

Kim, S. and S. Lee. 2009. “Improved intermediate performance of rateless codes”. In: *Advanced Communication Technology, 2009. ICACT 2009. 11th International Conference on*. Vol. 3. IEEE. 1682–1686.

Koetter, R. and M. Medard. 2003. “An algebraic approach to network coding”. *IEEE/ACM Transactions on Networking*. 11(5): 782–795. ISSN: 1063-6692. DOI: 10.1109/TNET.2003.818197.

Konstantinidis, K. and A. Ramamoorthy. 2018. “Leveraging Coding Techniques for Speeding up Distributed Computing”. *e-print arXiv:1802.03049*.

Korner, J. and K. Marton. 1979. “How to encode the modulo-two sum of binary sources”. *IEEE Transactions on Information Theory*. 25(2): 219–221.

Kosaian, J., K. Rashmi, and S. Venkataraman. 2018. “Learning a code: Machine learning for approximate non-linear coded computation”. *arXiv preprint arXiv:1806.01259*.

Krizhevsky, A. and G. Hinton. 2009. "Learning multiple layers of features from tiny images". *Tech. rep.* Citeseer.

Kushilevitz, E. and N. Nisan. 2006. *Communication Complexity*. Cambridge University Press.

Lee, K., M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran. 2018. "Speeding Up Distributed Machine Learning Using Codes". *IEEE Transactions on Information Theory*. 64(3): 1514–1529. ISSN: 0018-9448. DOI: 10.1109/TIT.2017.2736066.

Lee, K., C. Suh, and K. Ramchandran. 2017. "High-dimensional coded matrix multiplication". In: *2017 IEEE International Symposium on Information Theory (ISIT)*. 2418–2422. DOI: 10.1109/ISIT.2017.8006963.

Lee, K., R. Pedarsani, and K. Ramchandran. 2015. "On scheduling redundant requests with cancellation overheads". In: *53rd Annual Allerton Conference on Communication, Control, and Computing*. IEEE. 99–106.

Li, S., M. A. Maddah-Ali, and A. S. Avestimehr. 2018a. "Compressed Coded Distributed Computing". In: *IEEE International Symposium on Information Theory (ISIT)*.

Li, S., M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr. 2018b. "A Fundamental Tradeoff Between Computation and Communication in Distributed Computing". *IEEE Transactions on Information Theory*. 64(1). ISSN: 0018-9448. DOI: 10.1109/TIT.2017.2756959.

Li, S., S. M. M. Kalan, A. S. Avestimehr, and M. Soltanolkotabi. 2018c. "Near-Optimal Straggler Mitigation for Distributed Gradient Methods". *IPDPSW*. May.

Li, S., S. M. M. Kalan, Q. Yu, M. Soltanolkotabi, and A. S. Avestimehr. 2018d. "Polynomially Coded Regression: Optimal Straggler Mitigation via Data Encoding". *e-print arXiv:1805.09934*.

Li, S., M. A. Maddah-Ali, and A. S. Avestimehr. 2016a. "A Unified Coding Framework for Distributed Computing with Straggling Servers". *IEEE NetCod*. Dec.

Li, S., M. A. Maddah-Ali, and A. S. Avestimehr. 2015. "Coded MapReduce". *53rd Annual Allerton Conference on Communication, Control, and Computing*. Sept.

Li, S., M. A. Maddah-Ali, and A. S. Avestimehr. 2016b. “Coded Distributed Computing: Straggling Servers and Multistage Dataflows”. *54th Allerton Conference*. Sept.

Li, S., M. Yu, C.-S. Yang, A. S. Avestimehr, S. Kannan, and P. Viswanath. 2018e. “PolyShard: Coded Sharding Achieves Linearly Scaling Efficiency and Security Simultaneously”. arXiv: 1809.10361 [cs.CR].

Liberty, E. and S. W. Zucker. 2009. “The mailman algorithm: A note on matrix–vector multiplication”. *Information Processing Letters*. 109(3): 179–182.

Lin, S. and D. J. Costello. 2004. *Error control coding*. Pearson.

Lindell, Y. 2005. “Secure multiparty computation for privacy preserving data mining”. In: *Encyclopedia of Data Warehousing and Mining*. IGI Global. 1005–1009.

Low, Y., D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. 2012. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. *Proc. VLDB Endow.* 5(8): 716–727. ISSN: 2150-8097. DOI: 10.14778/2212351.2212354. URL: https://doi.org/10.14778/2212351.2212354.

Maddah-Ali, M. A. and U. Niesen. 2014a. “Decentralized coded caching attains order-optimal memory-rate tradeoff”. *IEEE/ACM Transactions on Networking*. Apr.

Maddah-Ali, M. A. and U. Niesen. 2014b. “Fundamental limits of caching”. *IEEE Transactions on Information Theory*. 60(5): 2856–2867.

Maity, R. K., A. S. Rawat, and A. Mazumdar. 2018. “Robust Gradient Descent via Moment Encoding with LDPC Codes”. *SysML Conference*.

Maleki, H., V. R. Cadambe, and S. A. Jafar. 2014. “Index Coding—An Interference Alignment Perspective”. *IEEE Transactions on Information Theory*. 60(9): 5402–5432.

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. “Pregel: a system for large-scale graph processing”. In: *Proceedings of the ACM SIGMOD International Conference on Management of data*. ACM. 135–146.

Mallick, A., M. Chaudhari, U. Sheth, G. Palanikumar, and G. Joshi. 2019. “Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication”. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. 3(3): 1–40.

McEliece, R. J. and D. V. Sarwate. 1981. “On sharing secrets and Reed-Solomon codes”. *Communications of the ACM*. 24(9): 583–584.

McMahan, H. B., E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. 2017. “Communication-efficient learning of deep networks from decentralized data”. In: *Int. Conf. on Artificial Int. and Stat. (AISTATS)*. 1273–1282.

Melis, L., C. Song, E. D. Cristofaro, and V. Shmatikov. 2019. “Exploiting Unintended Feature Leakage in Collaborative Learning”. *arXiv:1805.04049*.

Mitra, D. and L. Dolecek. 2019. “Patterned Erasure Correcting Codes for Low Storage-Overhead Blockchain Systems”. In: *2019 53rd Asilomar Conference on Signals, Systems, and Computers*. IEEE. 1734–1738.

Mohassel, P. and Y. Zhang. 2017. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. Vol. 00. 19–38. DOI: 10.1109/SP.2017.12. URL: doi.ieeecomputersociety.org/10.1109/SP.2017.12.

Narra, K. G., Z. Lin, M. Kiamari, S. Avestimehr, and M. Annavaram. 2019. “Slack Squeeze Coded Computing for Adaptive Straggler Mitigation”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC ’19*. Denver, Colorado: Association for Computing Machinery. ISBN: 9781450362290. DOI: 10.1145/3295500.3356170. URL: https://doi.org/10.1145/3295500.3356170.

Nazer, B. and M. Gastpar. 2007. “Computation over multiple-access channels”. *IEEE Transactions on Information Theory*. 53(10): 3498–3516.

Nodehi, H. A. and M. A. Maddah-Ali. 2018. “Limited-Sharing Multi-Party Computation for Massive Matrix Operations”. In: *IEEE International Symposium on Information Theory (ISIT)*. 1231–1235. DOI: 10.1109/ISIT.2018.8437651.

O'Malley, O. 2008. "TeraByte Sort on Apache Hadoop". *Tech. rep.* Yahoo.

"Open MPI: Open Source High Performance Computing". https://www.open-mpi.org/.

Orlitsky, A. and A. El Gamal. 1990. "Average and randomized communication complexity". *IEEE Transactions on Information Theory*. 36(1): 3–16. ISSN: 0018-9448. DOI: 10.1109/18.50368.

Orlitsky, A. and J. Roche. 2001. "Coding for computing". *IEEE Transactions on Information Theory*. 47(3): 903–917. ISSN: 0018-9448. DOI: 10.1109/18.915643.

Pawar, S., S. El Rouayheb, and K. Ramchandran. 2011. "Securing dynamic distributed storage systems against eavesdropping and adversarial attacks". *IEEE Transactions on Information Theory*. 57(10): 6734–6753.

Poulson, J., B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. 2013. "Elemental: A new framework for distributed memory dense matrix computations". *ACM Transactions on Mathematical Software*. 39(2): 13:1–13:24. DOI: 10.1145/2427023.2427030.

Prakash, S., A. Reisizadeh, R. Pedarsani, and S. Avestimehr. 2018. "Coded Computing for Distributed Graph Analytics". *IEEE ISIT*.

Rajaraman, A. and J. D. Ullman. 2011. *Mining of massive datasets*. Cambridge University Press.

Ramamoorthy, A. and M. Langberg. 2013. "Communicating the sum of sources over a network". *IEEE Journal on Selected Areas in Communications*. 31(4): 655–665.

Raviv, N., I. Tamo, R. Tandon, and A. G. Dimakis. 2017. "Gradient Coding from Cyclic MDS Codes and Expander Graphs". *e-print arXiv:1707.03858*.

Rawat, A. S., O. O. Koyluoglu, N. Silberstein, and S. Vishwanath. 2014. "Optimal locally repairable and secure codes for distributed storage systems". *IEEE Transactions on Information Theory*. 60(1): 212–236.

Recht, B., C. Re, S. Wright, and F. Niu. 2011. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in neural information processing systems (NIPS)*. 693–701.

Reisizadeh, A., S. Prakash, R. Pedarsani, and A. S. Avestimehr. 2019. "Coded Computation Over Heterogeneous Clusters". *IEEE Transactions on Information Theory*. 65(7): 4227–4242.

Reisizadeh, A., S. Prakash, R. Pedarsani, and S. Avestimehr. 2017. "Coded computation over heterogeneous clusters". *IEEE ISIT*: 2408–2412.

Renteln, P. 2013. *Manifolds, Tensors, and Forms: An Introduction for Mathematicians and Physicists*. Cambridge University Press.

Roth, R. 2006. *Introduction to coding theory*. Cambridge University Press.

Sahraei, S. and A. S. Avestimehr. 2019. "INTERPOL: Information theoretically verifiable polynomial evaluation". In: *2019 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 1112–1116.

Sahraei, S., M. A. Maddah-Ali, and S. Avestimehr. 2019. "Interactive Verifiable Polynomial Evaluation". *arXiv preprint arXiv:1907.04302*.

Sanghavi, S. 2007. "Intermediate performance of rateless codes". In: *Information Theory Workshop, 2007. ITW'07. IEEE*. IEEE. 478–482.

Schölkopf, B., R. Herbrich, and A. J. Smola. 2001. "A generalized representer theorem". In: *International conference on computational learning theory*. Springer. 416–426.

Seide, F., H. Fu, J. Droppo, G. Li, and D. Yu. 2014. "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns". *Fifteenth Annual Conference of the International Speech Communication Association*.

Shah, N. B., K. Lee, and K. Ramchandran. 2016. "When do redundant requests reduce latency?" *IEEE Transactions on Communications*. 64(2): 715–722.

Shah, N. B., K. Rashmi, and P. V. Kumar. 2011. "Information-theoretically secure regenerating codes for distributed storage". In: *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*. IEEE. 1–5.

Shamir, A. 1979a. “How to Share a Secret”. *Commun. ACM*. 22(11): 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: http://doi.acm.org/10.1145/359168.359176.

Shamir, A. 1979b. “How to share a secret”. *Communications of the ACM*. 22(11): 612–613.

Shanmugam, K., A. G. Dimakis, and M. Langberg. 2013. “Local graph coloring and index coding”. In: *2013 IEEE International Symposium on Information Theory*. 1152–1156.

Singleton, R. 1964. “Maximum distance q-nary codes”. *IEEE Transactions on Information Theory*. 10(2): 116–118.

So, J., B. Guler, and A. S. Avestimehr. 2020. “Turbo-Aggregate: Breaking the Quadratic Aggregation Barrier in Secure Federated Learning”. arXiv: 2002.04156 [cs.LG].

So, J., B. Guler, A. S. Avestimehr, and P. Mohassel. 2019. “CodedPrivateML: A Fast and Privacy-Preserving Framework for Distributed Machine Learning”. *CoRR*. abs/1902.00641. arXiv: 1902.00641. URL: http://arxiv.org/abs/1902.00641.

Song, L., C. Fragouli, and T. Zhao. 2017. “A pliable index coding approach to data shuffling”. *e-print arXiv:1701.05540*.

Tandon, R., Q. Lei, A. G. Dimakis, and N. Karampatziakis. 2017. “Gradient Coding: Avoiding Stragglers in Distributed Learning”. In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. *Proceedings of Machine Learning Research*. International Convention Centre, Sydney, Australia: PMLR. 3368–3376.

Tang, L., K. Konstantinidis, and A. Ramamoorthy. 2019. “Erasure Coding for Distributed Matrix Multiplication for Matrices With Bounded Entries”. *IEEE Communications Letters*. 23(1): 8–11.

“tc - show / manipulate traffic control settings”. http://lartc.org/manpages/tc.txt.

Ullman, J. D., A. V. Aho, and J. E. Hopcroft. 1974. “The design and analysis of computer algorithms”. *Addison-Wesley, Reading*. 4: 1–2.

Van De Geijn, R. A. and J. Watts. 1997. “SUMMA: Scalable universal matrix multiplication algorithm”. *Concurrency-Practice and Experience*. 9(4): 255–274.

Wan, K., D. Tuninetti, M. Ji, and P. Piantanida. 2018. “Fundamental Limits of Distributed Data Shuffling”. *e-print arXiv:1807.00056*.

Wang, D., G. Joshi, and G. Wornell. 2014. “Efficient task replication for fast response times in parallel computation”. In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 42. No. 1. ACM. 599–600.

Wang, H., Z. Charles, and D. Papailiopoulos. 2019a. “ErasureHead: Distributed gradient descent without delays using approximate gradient coding”. *arXiv preprint arXiv:1901.09671*.

Wang, S., J. Liu, and N. Shroff. 2018a. “Coded sparse matrix multiplication”. *e-print arXiv:1802.03430*.

Wang, S., J. Liu, and N. Shroff. 2019b. “Fundamental Limits of Approximate Gradient Coding”. *Proc. ACM Meas. Anal. Comput. Syst.* 3(3). DOI: 10.1145/3366700. URL: https://doi.org/10.1145/3366700.

Wang, S., J. Liu, N. Shroff, and P. Yang. 2018b. “Fundamental limits of coded linear transform”. *e-print arXiv:1804.09791*.

Wen, W., C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. 2017. “TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning”. *Advances in neural information processing systems (NIPS)*: 1508–1518.

Woolsey, N., R.-R. Chen, and M. Ji. 2018. “A New Combinatorial Design of Coded Distributed Computing”. *e-print arXiv:1802.03870*.

Yang, H. and J. Lee. 2019. “Secure Distributed Computing With Straggling Servers Using Polynomial Codes”. *IEEE Transactions on Information Forensics and Security*. 14(1): 141–150.

Yang, Y., M. Interlandi, P. Grover, S. Kar, S. Amizadeh, and M. Weimer. 2019. “Coded Elastic Computing”. In: *2019 IEEE International Symposium on Information Theory (ISIT)*. 2654–2658.

Yang, Y., P. Grover, and S. Kar. 2017. “Coded Distributed Computing for Inverse Problems”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc. 709–719. URL: http://papers.nips.cc/paper/6673-coded-distributed-computing-for-inverse-problems.pdf.

Yao, A. C.-C. 1979. “Some complexity questions related to distributive computing (preliminary report)”. In: *Proceedings of the eleventh annual ACM symposium on Theory of computing*. 209–213.

Ye, M. and E. Abbe. 2018. “Communication-Computation Efficient Gradient Coding”. *e-print arXiv:1802.03475*.

Yu, M., S. Sahraei, S. Li, S. Avestimehr, S. Kannan, and P. Viswanath. 2020. “Coded Merkle Tree: Solving Data Availability Attacks in Blockchains”. In: *Financial Cryptography and Data Security (FC)*.

Yu, Q., S. Li, M. A. Maddah-Ali, and A. S. Avestimehr. 2017a. “How to Optimally Allocate Resources for Coded Distributed Computing?” *IEEE International Conference on Communications (ICC)*. May: 1–7.

Yu, Q., M. A. Maddah-Ali, and A. S. Avestimehr. 2017b. “Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication”. *NIPS*: 4406–4416.

Yu, Q., M. A. Maddah-Ali, and A. S. Avestimehr. 2018a. “Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding”. *e-print arXiv:1801.07487*.

Yu, Q., M. A. Maddah-Ali, and A. S. Avestimehr. 2018b. “Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding”. In: *IEEE International Symposium on Information Theory (ISIT)*. 2022–2026.

Yu, Q., N. Raviv, J. So, and A. S. Avestimehr. 2018c. “Lagrange Coded Computing: Optimal Design for Resiliency, Security and Privacy”. *e-print arXiv:1806.00939*.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. “Spark: cluster computing with working sets”. In: *Proceedings of the 2nd USENIX HotCloud*. Vol. 10. 10.

Zaharia, M., A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. 2008. “Improving MapReduce Performance in Heterogeneous Environments”. *OSDI*. 8(4): 7.

Zhang, S., J. Han, Z. Liu, K. Wang, and S. Feng. 2009. “Accelerating MapReduce with distributed memory cache”. *15th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. Dec.: 472–478.

Zhang, Z., L. Cherkasova, and B. T. Loo. 2013. “Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments”. In: *IEEE Sixth International Conference on Cloud Computing*. 839–846.

Zhuang, Y., W.-S. Chin, Y.-C. Juan, and C.-J. Lin. 2013. “A fast parallel SGD for matrix factorization in shared memory systems”. In: *Proceedings of the 7th ACM conference on Recommender systems*. ACM. 249–256.