Speeding Up Distributed Machine Learning using Codes

Kangwook Lee$^{\epsilon}$, Maximilian Lam$^{\epsilon}$, Ramtin Pedarsani$^{\epsilon}$
Dimitris Papailiopoulos$^{\alpha,\epsilon}$ and Kannan Ramchandran$^{\epsilon}$
$^{\alpha}$AMPLab, $^{\epsilon}$EECS at UC Berkeley

Abstract

Codes are widely used in many engineering applications to offer some form of reliability and fault tolerance. The high-level idea of *coding* is to exploit resource *redundancy* to deliver higher *robustness* against *system noise*. In distributed systems, there are several types of "noise" that can affect ML algorithms: straggler nodes, system failures, communication bottlenecks, etc. Moreover, *redundancy* is abundant: a plethora of nodes, a lot of spare storage, etc. However, there has been little interaction between *Codes*, *Machine Learning*, and *Distributed Systems*.

In this work, we scratch the tip of the "Coding for Distributed ML" iceberg. We show how codes can be used to speed up two of the most basic building blocks of distributed ML algorithms: *data shuffling* and *matrix multiplication*. In data shuffling, we use codes to exploit the excess in storage and reduce communication bottlenecks. For matrix multiplication, we use codes to leverage the plethora of nodes and alleviate the effects of stragglers. We provide theoretical insights and evidence on synthetic and OpenMPI experiments on Amazon EC2 that highlight significant gains offered by *coded* solutions compared to uncoded ones.

1 Introduction

Codes have been used in a large range of engineering applications to obtain some form of performance reliability [2]. The central theme in coding is to exploit redundancy in sophisticated ways to counter the adverse effects of system noise. The notions of "redundancy", "noise", and "reliability" have different meanings in different contexts. For example in distributed storage, codes are used to generate redundant functions of the data to guarantee that even in the presence of machine failures, the lost data can be restored [4,7,9]. Although for many of these applications data replication can be thought of as a viable alternative, more often that not sophisticated codes *significantly* outperform replication with respect to many metrics of interest.

In recent years, deploying algorithms on distributed systems became a popular means to scale-up large ML tasks. However, the performance of these algorithms is significantly affected by anomalous system behavior or bottlenecks [3]; a form of "system noise". For example, algorithms that require passing large amounts of data among nodes can easily become *communication-bound*. At the same time, *straggler nodes* (i.e., nodes that are significantly slower than the average) cause bottlenecks and hinder the promised speedups of distributed implementations.

On the other hand, there is a surplus of *redundancy* to be exploited in these large systems. A large amount of storage and an abundance of nodes could be used to counter system anomalies and offer higher *performance reliability*. In this work, we explore a new research agenda, that is driven by the question: *Can codes speed up distributed machine learning?* We open up this new unexplored thread by providing substantial theoretical and experimental evidence where the use of codes significantly improves the performance of several distributed ML algorithms.

Where can we use codes? We first observe that we can abstract the distributed ML work-flow as operating on three distinct layers: a storage layer, a communication, and finally a local computation layer. Recently, codes have been extensively used for the storage layer to provide reliability by exploiting storage redundancy [4, 7, 9]. The idea there is simple: if a node is storing say x, another is storing y, and a third is storing $x + y$, then any single node failure leads to no data loss. Any two of the three nodes can recover both x and y.

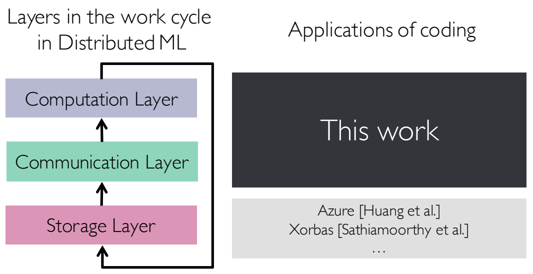

Figure 1: Distributed ML work-flow and applications of coding.

In this work, we steer away from storage and focus on the communication and computation layers. We embark on this trip by observing that many ML algorithms are based on two central building blocks (among others): *data shuffling* and *matrix multiplication*. Data shuffling is relevant to the communication layer, and matrix multiplication is relevant to the computation layer. In this work we show how coding can partially alleviate the communication bottleneck in data shuffling and the effects of straggler nodes in matrix multiplication. In the following, we summarize our contributions.

Coded Shuffling Training a statistical model usually requires several passes over the data. It is common practice for stochastic learning algorithms to randomly re-shuffle the data after each pass. Although challenging to analyze theoretically, data shuffling is well-known to lead to superior statistical performance in practice [8, 10]. Unfortunately, when we scale up to distributed setups, data shuffling can become a significant bottleneck, as it incurs heavy communication.

Using cues from caching solutions in content dissemination networks [6], we develop *Coded Shuffling*: a novel data-shuffling scheme that employs coding to reduce communication. *Coded Shuffling* stores and reuses *cached* information across different passes of the data to reduce communication. Theoretically we can show that coding can reduce communication during shuffling by at least a constant factor. In synthetic experiments on several distributed ML tasks, where show that *Coded Shuffling* out-performs uncoded, and no shuffling at all.

Coded Multiplication Matrix multiplication is an indispensable computational block for nearly all basic ML tools, from gradient computation and spectral analysis, to solving systems of equations. Moreover, it is well known to be “embarrassingly parallel” making it a perfect fit for distributed implementations. However, in distributed setups the performance of “embarrassingly parallel” tasks is hindered by the slowest nodes, i.e., the *stragglers*. Stragglers are caused by several factors such as system heterogeneity, node failures, communication delays, etc.

We present *Coded Multiplication*, a novel framework for parallel matrix multiplication that uses codes to alleviate the effects of straggler nodes. We use *erasure codes* to design parallel tasks so that the speedups of the algorithm are robust up to a certain number of stragglers. Interestingly, even protecting against a small number of stragglers can significantly boost the performance of distributed algorithms. We support *Coded Multiplication* by extensive benchmarking on Amazon EC2 and compare it against popular parallelization schemes used in practice.

2 Coded data shuffling

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

We illustrate the basics of *Coded Shuffling* with a toy example. Consider a system with two worker nodes and one master node. Assume that the master node holds the entire data set $\mathbf{A}$, a set of n datapoints. For the sake of this example, assume the data to be equipartitioned in 4 batches $\mathbf{A}_1, \dots, \mathbf{A}_4$. Assume that node W_1 already has $\mathbf{A}_1$ and $\mathbf{A}_2$ cached, and node W_2 has $\mathbf{A}_3$ and $\mathbf{A}_4$

cached. Moreover, assume that the sole objective of the master is to transmit to the first worker $\mathbf{A}_3$ and to the second $\mathbf{A}_4$. For this purpose, the master node can simply broadcast a *coded* message $\mathbf{A}_2 + \mathbf{A}_3$ to the worker nodes. Since node 1 has access to $\mathbf{A}_2$, it can subtract $\mathbf{A}_2$ from the received message $\mathbf{A}_2+\mathbf{A}_3$, and replace $\mathbf{A}_2$ with $\mathbf{A}_3$. Similarly, node 2 can replace $\mathbf{A}_3$ with $\mathbf{A}_2$. Compared to the naïve (or uncoded) shuffling scheme in which the master node transmits $\mathbf{A}_2$ and $\mathbf{A}_3$ separately, this new shuffling scheme can save 50% of the communication cost, speeding up the overall machine learning algorithm. This is true assuming that *broadcasting* a message to all workers is significantly cheaper than sending individual messages to each worker.

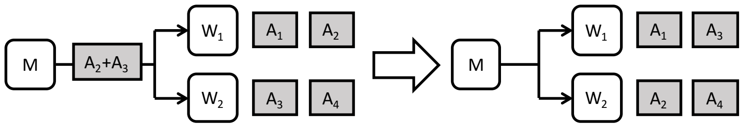

Figure 2: **Illustration of *Coded Shuffling*.** Data matrix $\mathbf{A}$ is partitioned into 4 submatrices: $\mathbf{A}_1$ to $\mathbf{A}_4$. Before shuffling, worker W_1 has $\mathbf{A}_1$ and $\mathbf{A}_2$ and worker W_2 has $\mathbf{A}_3$ and $\mathbf{A}_4$. The master node can send $\mathbf{A}_2 + \mathbf{A}_3$ in order to shuffle the data stored at the two workers.

Coded Shuffling is a generalization of the above toy example.

Theorem 1. *(informal) Let α be the fraction of the data matrix that can be cached at each worker, and K be the number of workers. Coded shuffling reduces the communication rate by a factor (approximately) $\frac{1}{\alpha K}$ compared to uncoded shuffling.*

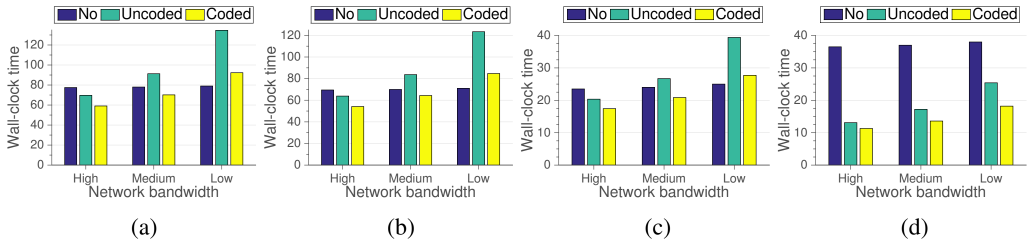

Figure 3: **Comparison of shuffling schemes.** We simulate Parallel-SGD for linear regression (LR) and classification (CL): (a) LR with synthetic data, (b) LR with real data, (c) CL with synthetic data, and (d) CL with real data. In most cases, we observe that the algorithm with *Coded Shuffling* achieves the best performance because it can shuffle data without incurring much communication cost. The algorithm with uncoded shuffling sometimes performs worse than the one without shuffling when the communication cost is too high compared to the statistical gain of shuffling.

We simulate the performance of parallel stochastic gradient descent algorithm [11] with various shuffling schemes: *Coded Shuffling*, *Uncoded Shuffling*, and *No Shuffling*. We apply the algorithm to solve linear regression and classification problems on synthetic and real data. We use wine quality data set for linear regression, and Spambase data set for classification, both from UCI Machine Learning Repository [1,5]. In order to account for the cost of communication, we consider the *wall-clock time*, the sum of computation time and communication time. We assume that computation of each epoch takes a unit time, and uncoded shuffling takes a half of a unit time, a unit time, or two units of time, respectively with high, medium, and low network bandwidth. Plotted in Figure 3 are the measured wall-clock times to achieve certain target errors. In all the results, coded shuffling achieves the best performance as it shuffles data without incurring heavy communication. Note that the algorithm with uncoded shuffling sometimes performs worse than the one without shuffling when the communication cost is too high compared to the statistical gain of shuffling.

3 Coded Multiplication

We now describe *Coded Multiplication* with the following illustrative example. Consider a system with three worker nodes and one master node, as depicted in Figure 4. The goal is to compute a matrix multiplication $\mathbf{AX}$. The data matrix $\mathbf{A}$ is vertically divided into two (equally tall) submatrices $\mathbf{A}_1$ and $\mathbf{A}_2$, which are stored in node 1 and node 2, respectively. In node 3, we store the sum of

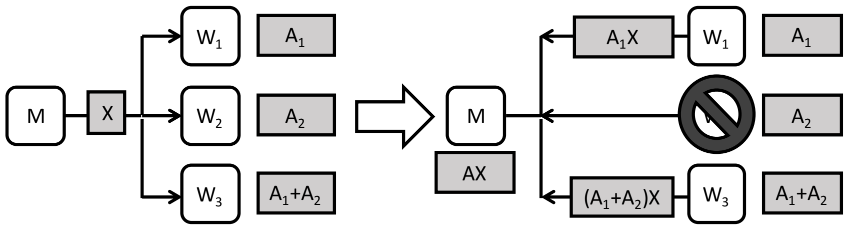

Figure 4: **Illustration of *Coded Multiplication*.** Data matrix $\mathbf{A}$ is partitioned into 2 submatrices: $\mathbf{A}_1$ and $\mathbf{A}_2$. Node W_1 stores $\mathbf{A}_1$, node W_2 stores $\mathbf{A}_2$, and node W_3 stores $\mathbf{A}_1 + \mathbf{A}_2$. Upon receiving $\mathbf{X}$, each node multiplies $\mathbf{X}$ with the stored matrix, and send the product to the master node. Observe that the master node can always recover $\mathbf{AX}$ upon receiving *any* 2 products, without needing to wait for the latest response.

the two submatrices $\mathbf{A}_1 + \mathbf{A}_2$. After the master node transmits $\mathbf{X}$ to the worker nodes, each node computes the matrix multiplication of the stored matrix and the received matrix $\mathbf{X}$, and sends the computation result back to the master node. The master node can compute $\mathbf{AX}$ as soon as it receives *any* two computation results. For instance, consider a case where it collects $\mathbf{A}_1\mathbf{X}$ from node 1 and $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$ from node 3. By subtracting $\mathbf{A}_1\mathbf{X}$ from $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{X}$, it can recover $\mathbf{A}_2\mathbf{X}$ and hence $\mathbf{AX}$, which is a vertical concatenation of $\mathbf{A}_1\mathbf{X}$ and $\mathbf{A}_2\mathbf{X}$.

Coded Multiplication designs parallel tasks using erasure codes such that its runtime is not affected by up to a certain number of straggler. Hence, the runtime of the algorithm can be significantly reduced compared to that of the other uncoded matrix multiplication algorithms.

Theorem 2. *(informal) If the runtime of each subtask has an exponential tail, the optimal coded matrix multiplication is $\Theta(\log(K))$ times faster than the uncoded matrix multiplication with K workers.*

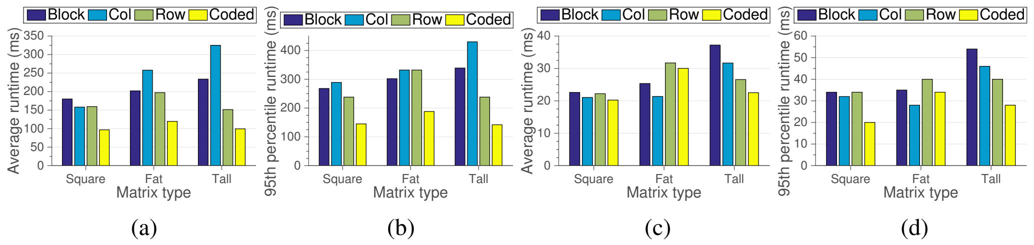

Figure 5: **Comparison of parallel matrix multiplication algorithms:** We compare various parallel matrix multiplication algorithms: block, column-partition, row-partition, and coded (row-partition) matrix multiplication. We implement the four algorithms using OpenMPI and test them on Amazon EC2 cluster of 25 instances. We measure the average and the 95th percentile runtime of the algorithms. Plotted in (a) and (b) are results with `m1-small` instances, and in (c) and (d) are with `c1-medium` instances.

We compare the performance of various parallel matrix multiplication algorithms. We implement three other popular parallel matrix multiplication algorithms and measure their performance on Amazon EC2 cluster of 25 instances. We randomly draw a square matrix of size 5750×5750, a fat matrix of size 5750×11500, and a tall matrix of size 11500×5750, and multiply them with a column vector. For *Coded Multiplication*, we design parallel tasks such that the runtime of the algorithm is not affected by any 2 stragglers. Figure 5 shows that *Coded Multiplication* outperforms all the other parallel matrix multiplication algorithms in most cases.

4 Conclusion

Can codes speed up distributed machine learning? In this work, we give an affirmative answer and make the first steps towards this novel and exciting direction. The produce of our preliminary work are *Coded Shuffling* and *Coded Multiplication*. Our proposed schemes show that *coding* can be used to speed up some of the most basic building blocks of distributed ML algorithms. Though we focused on specific applications of these techniques, our schemes can be generalized to a broader class of ML algorithms.

References

[1] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553, 2009.

[2] D Costello and Shu Lin. Error control coding. *New Jersey*, 2004.

[3] Jeffrey Dean and Luiz Andr Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[4] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, Boston, MA, 2012. USENIX.

[5] M. Lichman. UCI machine learning repository, 2013.

[6] Mohammad Ali Maddah-Ali and Urs Niesen. Fundamental limits of caching. *IEEE Transactions on Information Theory*, 60(5):2856–2867, 2014.

[7] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, Berkeley, CA, 2013. USENIX.

[8] Benjamin Recht and Christopher Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2):201–226, 2013.

[9] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 325–336. VLDB Endowment, 2013.

[10] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 7(12):1283–1294, 2014.

[11] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.