UberShuffle: Communication-efficient Data Shuffling for SGD via Coding Theory

Jichan Chung
EE at KAIST
jichan3751@kaist.ac.kr

Kangwook Lee
EE at KAIST
kw1jjang@kaist.ac.kr

Ramtin Pedarsani
ECE at UC Santa Barbara
ramtin@ece.ucsb.edu

Dimitris Papailiopoulos
ECE at University of Wisconsin-Madison
dimitris@papail.io

Kannan Ramchandran
EECS at UC Berkeley
kannanr@eecs.berkeley.edu

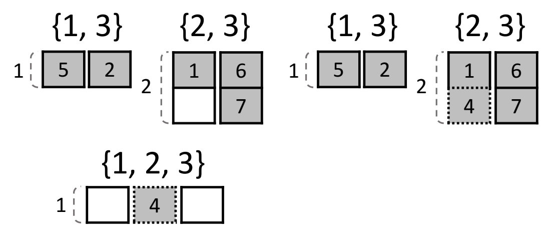
1 INTRODUCTION

Distributed machine learning systems are becoming increasingly popular due to their promise of high scalability and substantial speedup gains. In a prototypical distributed learning setup, each compute node computes gradient updates on parts of the dataset, and these updates are periodically synchronized at a parameter server. Recent works show that reshuffling the training data set across the compute nodes leads to superior convergence performance [3, 4, 6, 12, 13, 16, 17]. In practice, however, data shuffling incurs a large communication load and many practitioners avoid it.

One of the first *coded shuffling algorithms* for distributed machine learning was proposed in [8] to leverage the local caches of the compute nodes and curtail the communication load of the shuffling process. Based on a novel coding technique, the parameter server broadcasts linear combinations of data points, which are carefully designed such that every compute node can successfully decode its allocated batch. The authors show that such their proposed coded shuffling algorithm can –in theory– reduce the communication load by a factor of $\Theta(n)$, where the number of compute nodes is n. However, the practical efficacy of the coded shuffling algorithm has not been demonstrated yet. Indeed, the theoretical guarantees of [7, 8] hold only when the number of data points is approaching infinity.

The goal of this work is to exhibit that erasure coded algorithms for data shuffling can indeed lead to significant performance gains in practice. In this work, we present a new and implementable coded shuffling algorithm, called UberShuffle, based on [8]. We implement a distributed machine learning system that can run generic distributed machine learning algorithms combined with our shuffling procedure. We compare the performance of different shuffling algorithms under various setups, and show that the coded shuffling algorithms can achieve significant speed-up gains in practice. In some cases of our experiments, we observe that the data shuffling time is reduced by 47%, and the training time is reduced by 32%.

Related Works. When running distributed gradient descent algorithms, periodic shuffling of the training data under without-replacement sampling is observed to achieve large statistical gains [3, 4, 6, 11–14, 16, 17]. The coded shuffling algorithm is firstly proposed in [7, 8]. Several works have studied the fundamental limits of the shuffling problem and proposed various shuffling algorithms [1, 2, 15]. A similar coding idea is proposed for speeding up MapReduce applications [9, 10].



(a) Coded Shuffling (b) UberShuffle

Figure 1: Illustration of encoding table for Coded Shuffling algorithm and UberShuffle.

2 SYSTEM MODEL AND DATA SHUFFLING ALGORITHMS

Consider a distributed computing environment with n distributed compute nodes and a parameter server. Let us denote the compute node i by W_i and the parameter server by M. M has access to the entire data set, consisting of q (unit-sized) data points $(d_1, d_2, ..., d_q)$. At the beginning of each epoch, the system requires all of these data points to be randomly redistributed across n compute nodes. M draws a random assignment of each data point without replacement such that all W_is are assigned the same number of data points. We denote by D_i the set of data indices assigned to W_i. We also assume that each W_i can cache up to $s := \lfloor \alpha q \rfloor$ data points, for $1/q \le \alpha < 1$. We denote the set of data indices that is cached in W_i's cache by C_i. We implement data shuffling algorithms which performs the following task: each W_i recovers D_i from data transmissions from M and its locally stored data C_i, and updates C_i with the union of the set D_i and the set of $s - |D_i|$ data points randomly sampled (without replacement) from $C_i \setminus D_i$.

We breifly explain uncoded shuffling algorithm, coded shuffling algorithm [8] and UberShuffle using the following toy example. Consider a case where $n = 3$, $q = 9$, and $\alpha = 0.44$. Further, assume that $C_1 = \{2, 3, 4, 8\}, C_2 = \{6, 7, 8, 9\}, C_3 = \{1, 3, 4, 5\}$ and $D_1 = \{3, 5, 8\}, D_2 = \{1, 4, 9\}, D_3 = \{2, 6, 7\}$.

For the uncoded shuffling algorithm, M simply sends a set of data points $U_i := D_i \setminus C_i$ to each W_i. In this example, the algorithm will incur 6 transmission: $U_1 = \{5\}, U_2 = \{1, 4\}, U_3 = \{2, 6, 7\}$.

2.1 Coded Shuffling Algorithm

See Fig. 1a for visualization. Assume that M *broadcasts* $d_2 + d_5$. Since d_2 is stored in W_1 and d_5 is stored in W_3, W_1 can obtain d_5 by subtracting d_2 from $d_2 + d_5$, and W_3 can obtain d_5 similarly. One can visualize this example in the table labeled as $\{1, 3\}$ in Fig. 1a.

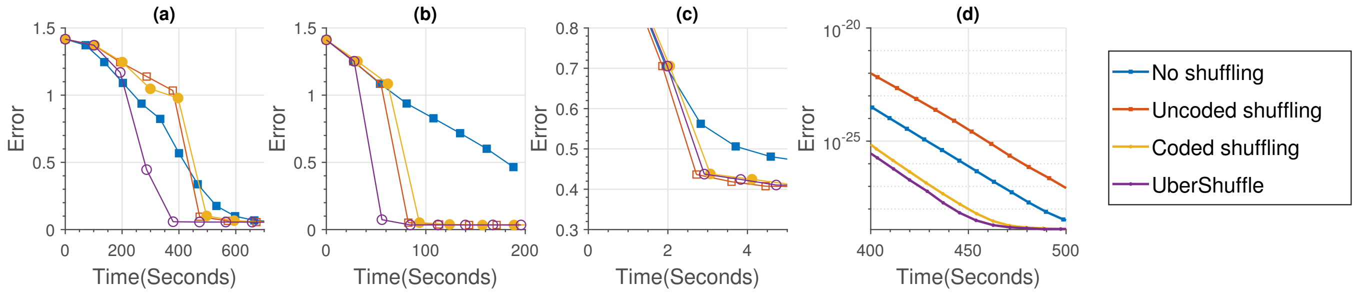


Figure 2: Convergence performances. (a) and (b) are for the matrix completion with synthetic data; (c) is for the matrix completion with real data; and (d) is for the linear regression with synthetic data.

The first column of the table corresponds to the data points that are required by W_1 and exclusively cached in $W_{\{1,3\}\setminus\{1\}}$, and the second column is for the data points required by W_3 and exclusively cached in $W_{\{1,3\}\setminus\{3\}}$. By finding the set of nodes where the data point is exclusively cached for each data point, one can obtain all the encoding tables shown in Fig. 1a. Once the encoding table is obtained, M simply generates one encoded packet per row of the encoding tables, and broadcasts them to the W_is. Here, note that the communication load of the coded shuffling algorithm for this example is 4 transmissions.

2.2 UberShuffle

See Fig. 1b for visualization. Recall that in Fig. 1a, there are a few missing entries in the encoding tables. The key idea of UberShuffle is to fill these gaps by reallocating data points between the encoding tables in order to reduce the number of packets. For instance, consider d_4. The original coded shuffling algorithm assigns this data point to the encoding table $\{1, 2, 3\}$ since d_4 is exclusively stored in $\{1, 3\}$ and required by W_2. While this can maximize the coding gain in the asymptotic regime, in this example with a finite number of data points, this may not be the optimal choice. For instance, one can reallocate d_4 to the first column of the encoding table $\{2, 3\}$, without compromising the decoding conditions. As a result of the reallocation, the communication load can be reduced from 4 to 3.

Roughly, the UberShuffle algorithm constructs a directed acyclic graph (DAG) representing flows corresponding to possible packet reallocations, and greedily optimizes the number of encoded packets by considering each layer of the DAG.

3 EXPERIMENTAL RESULTS

We implement a generic distributed machine learning system with various shuffling algorithms using Open MPI C, and evaluate the performance of our shufflng algorithms for the distributed SGD algorithm for a low-rank matrix completion problem, proposed in [12], on both synthetic data and real data. For the synthetic data, we generate low-rank matrices with random Gaussian factor matrices. For the real data, we use the Movielens 20m dataset [5], preprocessed by randomly sampling 68 data points from each row of the users, resulting in a sampled dataset with observation matrix of size $n_r \times n_c$ where $n_r = 68682$, $n_c = 16622$. We also run the parallel SGD (PSGD) algorithm [17] for linear regression with randomly generated synthetic data matrix A of size $q \times m$ and a vector x of size m, and vector y generated by $y = Ax$.

All experiments are run on an Amazon EC2 cluster with single `m3.2xlarge` (2.5GHz Intel Xeon E5-2670 v2 with 30GB RAM) instance for M, using 2 cores out of 8 cores, and 20 `m3.xlarge` (2.5Ghz Intel Xeon E5-2670 v2 with 15GB RAM) instances for W_is, each using 2 cores out of 4 cores.

Our experimental results are summarized in Table. 1 and Fig. 2. In Table 1, we observe that UberShuffle achieves the fastest shuffling times (including all the extra computational overheads) in most cases. In Fig. 2, the best performance is observed with UberShuffle for scenarios (a) and (b), and the convergence time is reduced by at least 19.8% and 32.1%, respectively.

We also implement data shuffling via shared storage system where computing nodes can directly access the new data points every epoch without relying parameter server's shuffling mechanism, considering two available options on Amazon Web Service: Elastic Block Storage (EBS) and Elastic File System (EFS). We compare the performances of these systems with UberShuffle on low-rank matrix completion ($n_r = n_c = 300k, r = 10, m = 1000, \alpha = 0.05$). As a result, we observe: $T_{\text{EBS}} = 110$, $T_{\text{EFS(general)}} = 490$, $T_{\text{EFS(I/O)}} = 1100$, and $T_{\text{UberShuffle}} = 34$, all in seconds. Thus, the UberShuffle algorithm is 3.2 times faster than the fastest storage-based alternative.

Table 1: Experimental setups and shuffling time comparison. 'CS = coded shuffling', 'US = UberShuffle', and 'UN = uncoded shuffling'; 'MC = matrix completion' and 'LR = linear regression'.

	q	m	α	Shuffling Time (sec)				Setup
				UN	CS	US	(CS-US)/CS	
(a)	10^5	10^4	0.2	105.5	123.6	**65.2**	47.2%	MC, Synthetic
(b)	3×10^5	1000	0.2	35.0	28.8	**25.3**	12.2%	MC, Synthetic
(c)	6.8×10^4	68	0.2	**0.42**	1.66	1.52	8.4%	MC, Real
(d)	7×10^5	2×10^5	0.14	112.90	94.60	**60.82**	35.7%	LR, Synthetic

REFERENCES

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

[2] M. A. Attia and R. Tandon. 2016. On the worst-case communication overhead for distributed data shuffling. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 961–968. https://doi.org/10.1109/ALLERTON.2016.7852338

[3] Léon Bottou. 2012. Stochastic Gradient Descent Tricks. In *Neural Networks: Tricks of the Trade - Second Edition*. 421–436. https://doi.org/10.1007/978-3-642-35289-8_25

[4] Mert Gürbüzbalaban, Asu Ozdaglar, and Pablo Parrilo. 2015. Why Random Reshuffling Beats Stochastic Gradient Descent. *arXiv preprint arXiv:1510.08560* (2015).

[5] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (Dec. 2015), 19 pages. https://doi.org/10.1145/2827872

[6] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[7] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2015. Speeding up distributed machine learning using codes. In *the Workshop on ML Systems at NIPS*.

[8] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2017. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory* (2017).

[9] Songze Li, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. 2015. Coded MapReduce. In *Communication, Control, and Computing (Allerton), 2015 53rd Annual Allerton Conference on*. IEEE, 964–971.

[10] Songze Li, Sucha Supittayapornpong, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. 2017. Coded terasort. *arXiv preprint arXiv:1702.04850* (2017).

[11] B. Recht and C. Re. 2012. Beneath the valley of the noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences. *ArXiv e-prints* (Feb. 2012). arXiv:math.OC/1202.4184

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

[13] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proc. of the 25th Annual Conference on Neural Information Processing (NIPS)*. 693–701.

[14] Ohad Shamir. 2016. Without-Replacement Sampling for Stochastic Gradient Methods. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 46–54. http://papers.nips.cc/paper/6245-without-replacement-sampling-for-stochastic-gradient-methods.pdf

[15] L. Song, C. Fragouli, and T. Zhao. 2017. A pliable index coding approach to data shuffling. In *2017 IEEE International Symposium on Information Theory (ISIT)*. 2558–2562. https://doi.org/10.1109/ISIT.2017.8006991

[16] Ce Zhang and Christopher Re. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB* 7, 12 (2014), 1283–1294. http://www.vldb.org/pvldb/vol7/p1283-zhang.pdf

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