UberShuffle: Faster Distributed Learning via Erasure Coded Data Shuffling

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

Abstract

Modern large-scale learning algorithms are deployed on hundreds of distributed compute instances, each computing gradient updates on a subset of the training data. It has been empirically observed that these algorithms can offer higher statistical performance when the training data is *shuffled* once every few epochs. However, *data shuffling* is often avoided due to its heavy communication costs. Recently, erasure coding ideas have been proposed to minimize the communication cost of shuffling, but have been only successful in theory. In this work, we implement *UberShuffle*, a new erasure coded shuffling system and compare its performance against various shuffling algorithms in distributed learning setups. We observe that our shuffling framework for machine learning can achieve significant speed-ups compared to the state of the art. In some cases, the data shuffling time is reduced by about 50%, and the training time is reduced by about 30%.

1 Introduction

Distributed machine learning systems are becoming increasingly popular due to their promise of high scalability and substantial speedups gains. In a prototypical distributed learning setup, each worker computes gradient updates on parts of the dataset, and these updates are periodically synchronized at a master node (*i.e.,* parameter server). Recent works show that if the training data set is reshuffled across the worker nodes, this can lead to superior convergence performance [11, 14]. In practice, however, data shuffling incurs a huge communication cost and many practitioners avoid it.

The first *coded shuffling algorithm* was proposed in [7] to leverage the local caches of the worker nodes and curtail the communication cost of the shuffling process. Based on a novel coding technique, the master node broadcasts linear combinations of data points, which are carefully designed such that every worker can successfully decode its allocated batch. The authors show that such coded algorithm can –in theory– reduce the communication overhead by a factor of $\Theta(n)$, where the number of workers is n. However, the practical efficacy of the coded shuffling algorithm has not been demonstrated yet. Indeed, the theoretical guarantees of [6, 7] hold only when 1) the number of data points is approaching infinity, and when 2) there exists a perfect broadcast channel between the master node and the workers.[1] 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 the original work of [7]. We implement a distributed machine learning system that can run generic distributed machine

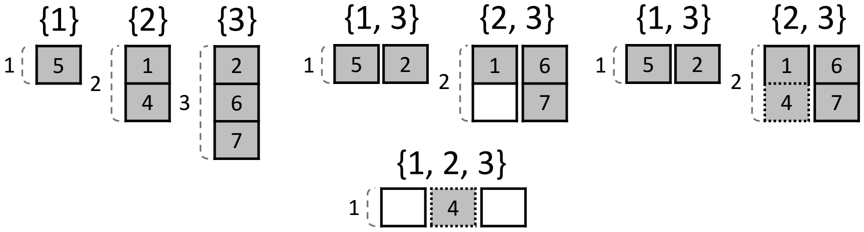
[1] In [7], the authors show that one can achieve the reduction factor of $\Theta(n/\log n)$ without broadcast channel.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

learning algorithms combined with any shuffling procedure. Using this system, 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.

Related Works The coded shuffling algorithm is firstly proposed in [6, 7]. Several works have studied the fundamental limits of the shuffling problem and proposed various shuffling algorithms [1, 2, 13]. A similar coding idea is proposed for speeding up MapReduce applications [8, 9].

2 System Model and Data Shuffling Algorithms



(a) Uncoded Shuffling (b) Coded Shuffling (c) UberShuffle

Figure 1: Illustration of encoding table for each shuffling Algorithm.

Consider a master-worker distributed computing environment n distributed workers and a master. The master has access to the entire data set, consisting of q (unit-sized) data points $(d_1, d_2, ..., d_q)$, and each worker can cache up to $s := \lfloor \alpha q \rfloor$ data points, for $1/q \le \alpha < 1$. [2] At the beginning of each epoch, the system requires all of these data points to be randomly redistributed across n workers. The master node first draws a random assignment of each data point such that all workers are assigned the same number of data points. Let us denote the destination of data point d_i by $w_i \in \{1, 2, \cdots, n\}$. We denote by D_i the set of data indices assigned to worker i, i.e., $D_i = \{j \mid w_j = i\}$, and denote by C_i the set of data indices that is cached in worker i's cache. We now provide informal illustrations of the shuffling algorithms, proposed in [7].

2.1 The uncoded shuffling algorithm

For each i, the master node unicasts to worker i the set of data points in $D_i \setminus C_i$, i.e., the data points that are required by worker i but are not cached in it.

Example 1 (The uncoded shuffling algorithm). 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 $D_1 = \{3, 5, 8\}, D_2 = \{1, 4, 9\}, D_3 = \{2, 6, 7\}$. The uncoded shuffling algorithm will incur 6 unicasts: d_5 to worker 1, d_1, d_4 to worker 2, and d_2, d_6, d_7 to worker 3. See Fig. 1a for visualization.

In general, the communication cost of the uncoded shuffling algorithm is $q(1 - s/q) = q(1 - \alpha)$ in terms of the number of unicasts per epoch [6].

2.2 The coded shuffling algorithm

The key idea of the coded shuffling algorithm is simple: instead of transmitting data points one by one, the master node linearly combines multiple data points and broadcasts a fewer number of *coded* data points. The coded shuffling algorithm judiciously encodes the data points such that each worker can obtain new data points by decoding the received message with the aid of cached data points.

Example 2 (The coded shuffling algorithm). We consider the same scenario considered in the previous example. Consider data point d_5, which is required by worker 1 for the subsequent epoch of the learning algorithm. Observe that this data point is *exclusively* cached in worker 3. On the other

[2] When $\alpha = 1$, every node can store the entire dataset, and hence shuffling is not needed anymore. However, this is not feasible if the dataset is too large to fit in a single memory. (For instance, the size of Google Books Ngram is 2.2TB [10].) One may achieve $\alpha = 1$ by storing the entire dataset in large storage units (such as SSD/HDD) but this approach is either cost-inefficient or slow due to excessive storage I/O overhead.

hand, the data point d_2 is required by worker 3, and it is *exclusively* cached in worker 1. Assume that master node *broadcasts* $d_2 + d_5$ to worker 1 and 3. Worker 1 can obtain d_5 by subtracting d_2 from $d_2 + d_5$, and worker 3 can obtain d_5 similarly. Hence, the master node can reallocate two data points using a single broadcast transmission, reducing the communication overheads. Let us denote a set of data points that are required by worker i, and exclusively cached by the workers in $\mathcal{I} \setminus \{i\}$ by $Z(\mathcal{I}, i)$. One can visualize this indexing result in the table labeled as $\{1, 3\}$ in Fig. 1b. The first column of the table corresponds to the set $Z(\{1, 3\}, 1)$ and the second column is for $Z(\{1, 3\}, 3)$. By repeating this procedure for every data point, one can obtain all the encoding tables shown in Fig. 1b. Once the encoding table is obtained, the master node simply generates one encoded packet per row of the encoding tables, and broadcasts them to the workers. By the property of the indexing procedure, every worker is guaranteed to obtain the desired data points by decoding the received packets. Here, note that the total communication cost of the coded shuffling algorithm for this example is 4 transmissions.

The communication cost of the coded shuffling algorithm is characterized in [7].

Theorem 1 ([7] The coded shuffling algorithm). *As q approaches infinity, the number of broadcasts of the coded shuffling algorithm is $R_c = \frac{q}{(np)^2}\left((1-p)^{n+1} + (n-1)p(1-p) - (1-p)^2\right)$ per epoch, where $p = \frac{s-q/n}{q-q/n}$.*

This theorem implies that as q gets large, and n grows sublinearly in q, $R_c \to \frac{q(1-\alpha)}{\alpha n}$. Thus, assuming a broadcast channel of bandwidth B between the master node and the workers and denoting the shuffling time by T, we can characterize the shuffling times as follows.

Corollary 2 (Shuffling times with broadcasts). *The shuffling time of the coded shuffling algorithm is $T_{\text{broadcast, coded}} \simeq \frac{q}{B}\frac{(1-\alpha)}{\alpha n}$. Similarly, the shuffling time of the uncoded shuffling algorithm is $T_{\text{broadcast, uncoded}} = \frac{q}{B}(1-\alpha)$, and hence $T_{\text{broadcast, coded}} \simeq \frac{1}{\alpha n} T_{\text{uncoded}}$.*

3 Coded Shuffling in Practice

The coded shuffling algorithm cannot be immediately deployed in practice due to a few limitations. First, it assumes a perfect broadcast channel between the master and the workers, which is not available in many practical applications. Further, the theoretical guarantee holds only when the number of data points approaches infinity, and its performance with a finite number of data points is unknown. In this section, we propose a few modification to the coded shuffling algorithm to address these limitations, making the algorithm more applicable in practice.

3.1 Van de Gejin's pseudo-broadcast algorithm

In [6], it is assumed that the master can transmit a packet to all n workers in one time slot. However, in practice, nodes in clusters are usually connected to network via (bidirectional) single-ported network interfaces, i.e., at most one message can be sent to or received from another node at full bandwidth. Hence, a natural question is whether one can still achieve the promised gain of the coded shuffling algorithm under such a practical setup. It turns out that under a mild assumption, one can make use of Van de Gejin's pseudo-broadcast algorithm [3], which fully exploits interconnections between the workers, to achieve the gain of the coded shuffling algorithm.

In [3], Van de Gejin et al. propose the following pseudo-broadcast algorithm. (See Fig. 2 for visual illustration.) In stage 1, the master node divides the packet into n subpackets and scatters them: the i^{th} packet is sent to worker i for $1 \le i \le n$. In the subsequent stages, the n workers circulate the subpackets, forming a ring structure. More precisely, for $1 \le j \le n-1$, in stage $j+1$, worker i transmits $\{((i+j-2) \bmod n) + 1\}^{\text{th}}$ subpacket to worker $\{(i \bmod n) + 1\}$. The time taken for the first stage of the algorithm is $n \times \frac{1}{n} T_{\text{unicast}} = T_{\text{unicast}}$. The subsequent stages can be completed in time $(n-1) \times \frac{1}{n} T_{\text{unicast}}$ since transmissions of each subpacket can be run in parallel. Hence, for large n, one can acheive $T_{\text{mutlicast}} = \frac{2n-1}{n} T_{\text{unicast}} \simeq 2T_{\text{unicast}}$. Thus, assuming a fully connected mesh network of link bandwidth B between the nodes, we can characterize the shuffling times for the multicast environment as follows.

Theorem 3 (Shuffling times with multicasts). *The shuffling time of the coded shuffling algorithm with multicast channel is $T_{\text{multicast,coded}} = \frac{q}{B}\frac{(1-\alpha)}{\alpha n}\frac{2n-1}{n} \simeq \frac{q}{B}\frac{2(1-\alpha)}{\alpha n}$. Hence, $T_{\text{multicast,coded}} \simeq 2T_{\text{broadcast,coded}} = \frac{2}{\alpha n} T_{\text{uncoded}}$.*

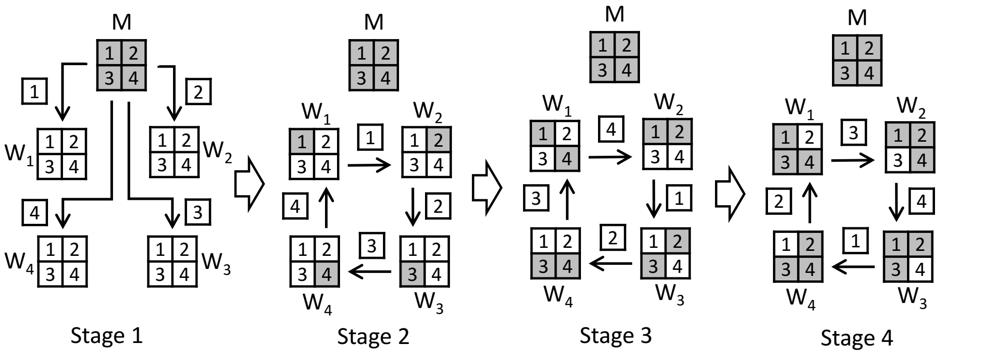


Figure 2: Illustration of Van de Gejin's pseudo-broadcast algorithm for $n = 4$. The packet is divided into n subpackets, indicated by square boxes. Dark boxes indicate that corresponding subpackets are already received.

3.2 UberShuffle: The coded shuffling algorithm with carpool

Thm. 1 makes an implicit assumption that the number of data points q grows faster than a certain growth rate. Recall that the master node generates one encoded packet per row of the encoding tables. If q grows fast enough, the number of allocated packets in every column of the table is the same. However, if q is finite, the number of allocated packets in the columns of the table are different from each other, and this results in resource wastage. More rigorously, one can show that the maximum number of packets allocated to any column of the table is almost equal to the average number of packets allocated to any column of the table if $q = \omega(e^n)$.[3] To improve the performance of the coded shuffling algorithm for finite q, we propose a variation of it, which we call UberShuffle.

Example 3 (UberShuffle). See Fig. 1c for visualization. Recall that in Fig. 1b, 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. This data point is exclusively stored in $\{1, 3\}$ and required by worker 2. The original coded shuffling algorithm assigns this data point to the encoding table $\{1, 2, 3\}$ since this can maximize the coding gain in the asymptotic regime. However, 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 $\{2, 3\}$. Note that such reallocation violates the 'exclusive' requirement of the original shuffling algorithm but does not compromise the decoding conditions. As a result of the reallocation, the total number of broadcast packets can be reduced from 4 to 3.

In general, the UberShuffle algorithm finds such reallocations between the encoding tables to fill the missing slots in them. (And this is why the algorithm is named 'UberShuffle': it resembles the famous carpool matching system.) Roughly, the UberShuffle algorithm first constructs a directed acyclic graph (DAG) between the columns of the encoding tables. In this DAG, a directed edge from column i of table A to column j of table B exists if and only if $i = j$ and $A \subset B$. (For instance, in the previous example, a directed edge exists from column 1 of table $\{1, 2, 3\}$ to column 1 of table $\{1, 3\}$.) After constructing the DAG, our algorithm finds a flow, which is corresponding to the packet reallocations, on the DAG to reduce the total number of encoded packets. More specifically, it greedily optimizes the number of encoded packets by considering each layer of the DAG.

To observe the gain of UberShuffle, we simulate the number of broadcast transmissions of various shuffling algorithms. Shown in Fig. 3 are simulation results with $n = 20$ and $q \in \{10^4, 10^5, 10^6\}$. One can observe that even with 10^6 data points ($q = 10^6$), the performance of the original coded shuffling algorithm is far from its theoretical guarantees (Thm. 1), and this gap is even larger as q decreases. Further, observe that the UberShuffle algorithm significantly outperforms the original coded shuffling algorithm. For instance, the UberShuffle algorithm is observed to reduce the number of packets by a factor of 5.4 when $q = 10^6$ and $\alpha = 0.55$ and by a factor of 2.58 when $q = 10^5$ and $\alpha = 0.325$.

[3] A similar phenomenon (though not identical) is also observed in the coded caching literature. (See [12].)

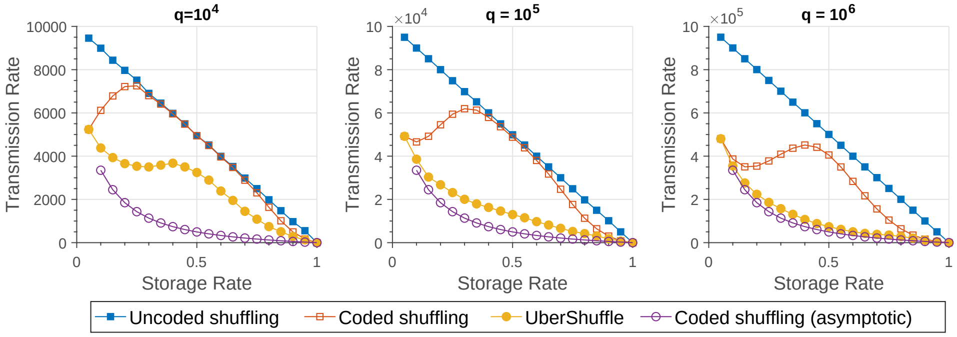


Figure 3: Number of packets used in Coded Shuffling algorithm for $n = 20$. Storage rate q/s is denoted by α.

4 Experimental Results

In this section, we briefly introduce our distributed learning system equipped with various shuffling algorithms. We then evaluate the performances of various shuffling algorithms on several applications.

4.1 System design and machine learning algorithms

System design and cluster setup We implement a generic distributed machine learning system using Open MPI C. Further, we implement various shuffling algorithms in our system so that we can compare the performances of different shuffling algorithms.[4] We remark that our system is inherently fault-tolerant: when nodes fail or straggle, the master node ignore them and proceed to the next iteration with the other nodes.[5] All experiments are run on an Amazon EC2 cluster with the following configuration. We use a `m3.2xlarge` (8-core 2.5GHz Intel Xeon E5-2670 v2 with 30GB RAM) instance for the master, and 20 `m3.xlarge` (4-core 2.5Ghz Intel Xeon E5-2670 v2 with 15GB RAM) instances for the workers. Note that the master is configured to have a sufficiently large RAM since it has to hold the entire training dataset. Further, though it has 8 cores, we use only 2 cores for our experiments. Also, the workers are configured to have maximum network bandwidth, and to use only 2 cores as well.

Distributed SGD for Matrix Completion We evaluate the performance of our shufflng algorithms for the distributed SGD algorithm for a low-rank matrix completion problem, proposed in [11]. At a high level, the algorithm finds the low-rank completion of the observation matrix X of size $n_r \times n_c$ and rank r, which is partially observed. In [11], the authors propose a way of distributing the observed entries of X across workers and updating the parameters in parallel without incurring any conflicts. Further, the authors show that shuffling improves the convergence speed, and hence our shuffling algorithm can be used to reduce the communication cost of this procedure. For the experiments, we run our system 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]. We preprocess the dataset by randomly sampling 68 data points from each row of the uses, resulting in a sampled dataset with $n_r = 68682$, $n_c = 16622$, and $m = 68$.

Parallel SGD for Linear Regression We also run the parallel SGD (PSGD) algorithm [14] for linear regression, described as follows. The master node randomly initialize x, and then broadcasts it to all workers. At the same time, the master node reads the data matrix A and shuffles partial row vector a_i's of A and the corresponding y_i's to workers. Once the data rows are shuffled, the workers independently run the SGD algorithm using the local data rows. After all the data points are used to update x in each worker nodes, the master gathers them and computes the global parameter x by averaging. For data, we randomly generate a synthetic data matrix A of size $q \times m$ and a vector x of size m, and y is generated by $y = Ax$.

[4] Note that our coding schemes are not applied to other communication patterns such as model synchronization. However, if the size of model parameters is large, those communication overheads are not negligible anymore, and whether or not it can be reduced via similar coding techniques is an interesting open problem.

[5] Indeed, this approach is observed to converge well or even faster [4].

4.2 Experimental Results

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

Our experimental results are summarized in Table 1. We observe that UberShuffle achieves the fastest shuffling times (including all the extra computational overheads) in most cases. For scenario (c), the uncoded shuffling time achieves the best performance. This is because the cost of communicating data points is low compared to the computational overhead of coded shuffling algorithms.

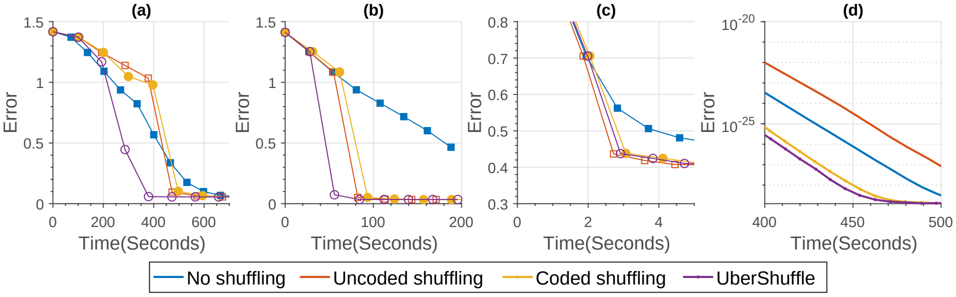


Figure 4: Convergence performances. (a) and (b) are for the matrix completion with synthetic data; (c) is for the matrix completion with real data; and (d) is for the linear regression with synthetic data.

We also report the convergence performance of the learning algorithms in Fig. 4. For scenarios (a) and (b), the best performance is observed with UberShuffle, and the convergence time is reduced by at least 19.8% and 32.1%, respectively. For scenario (c), while all of the shuffling algorithms outperform the one without shuffling, the coded shuffling algorithms indeed perform slightly worse than the uncoded shuffling algorithm. As discussed earlier, this is due to the relatively large computational overheads of the coded shuffling algorithms compared to the very low communication costs. For scenario (d), both the coded shuffling algorithms significantly outperform the other algorithms in terms of convergence time. Interestingly, the uncoded shuffling algorithms fails to perform better than the no shuffling algorithm due to its excessive communication costs.

Under our experimental setup where nodes are connected via 1Gbps bandwidth ethernet, transmission time is observed to be the dominant factor of the shuffling time, and hence we do not attempt at minimizing UberShuffle’s computational overhead (indexing/encoding/decoding time), which is negligible anyway. However, if much faster broadcast environment (such as RoCE with 50Gbps bandwidth) is available, this may not be the case anymore, and the shuffling gain might not be observed unless one minimizes the computational overhead of UberShuffle algorithm.

4.3 Data shuffling via shared storage systems

When a large enough shared storage system is available, one may store the entire dataset in it and let the workers directly access the new data points every epoch, without relying on the master node’s shuffling mechanism. Such a storage-based shuffling system is subject to network and disk bottlenecks, making itself less efficient than our shuffling system. To see this, we evaluate the performances of various shuffling algorithms on low-rank matrix completion ($n_r = n_c = 300k, r = 10, m = 1000, \alpha = 0.05$). We consider two available shared storage systems on Amazon Web Service: Elastic Block Storage (EBS) and Elastic File System (EFS). 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.

References

[1] M. A. Attia and R. Tandon. Information theoretic limits of data shuffling for distributed learning. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2016.

[2] M. A. Attia and R. Tandon. On the worst-case communication overhead for distributed data shuffling. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 961–968, Sept 2016.

[3] Mike Barnett, Lance Shuler, Robert van De Geijn, Satya Gupta, David G Payne, and Jerrell Watts. Interprocessor collective communication library (intercom). In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 357–364. IEEE, 1994.

[4] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. *ArXiv e-prints*, April 2016.

[5] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

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

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

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

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

[10] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K. Gray, The Google Books Team, Joseph P. Pickett, Dale Holberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin A. Nowak, and Erez Lieberman Aiden. Quantitative analysis of culture using millions of digitized books. *Science*, 2010.

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

[12] K. Shanmugam, M. Ji, A. M. Tulino, J. Llorca, and A. G. Dimakis. Finite-length analysis of caching-aided coded multicasting. *IEEE Transactions on Information Theory*, 62(10):5524–5537, Oct 2016.

[13] L. Song, C. Fragouli, and T. Zhao. A pliable index coding approach to data shuffling. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 2558–2562, June 2017.

[14] 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.