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

Jichan Chung$^{\epsilon}$, Kangwook Lee$^{\epsilon}$, Ramtin Pedarsani$^{\alpha}$,
Dimitris Papailiopoulos$^{\beta}$ and Kannan Ramchandran$^{\gamma}$
$^{\epsilon}$EE at KAIST, $^{\alpha}$ECE at UC Santa Barbara,
$^{\beta}$ECE at University of Wisconsin-Madison, $^{\sigma}$EECS at UC Berkeley [*]

December 16, 2017

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 better 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, coding-theoretic ideas have been proposed to minimize the communication cost of shuffling. In this work, we implement UBERSHUFFLE, a new coded shuffling system. 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 47%, and the training time is reduced by 32%.

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 [1–7]. In practice, however, data shuffling incurs a large communication cost 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 cost 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 overhead 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 [8, 9] 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.

1.1 Related Works

1.1.1 Distributed Machine Learning and Data Shuffling

Distributed gradient decent methods on networked systems have been extensively studied in the literature [10–20]. When running distributed gradient descent algorithms, periodic shuffling of the training data is observed to achieve large statistical gains [1–7]. It has been experimentally demonstrated that parallel stochastic gradient descent algorithm converges faster under without-replacement sampling as opposed to with-replacement sampling [21, 22]. Our shuffling algorithm can be viewed as a cost-efficient enabler of the without-replacement sampling across the distributed nodes.

[*]Emails: jichan3751@kaist.ac.kr, kw1jjang@kaist.ac.kr, ramtin@ece.ucsb.edu, dimitris@papail.io, kannanr@eecs.berkeley.edu.

1.1.2 Coded Shuffling for Distributed Machine Learning

Indeed, our work falls under the umbrella of a recent paradigm of 'codes for distributed machine learning', proposed in [8, 9]. In the aforementioned work, the authors propose two different algorithms based on coding-theoretic ideas, called 'coded shuffling' and 'coded computation'. *The coded shuffling* algorithm alleviates communication bottlenecks of distributed computing networks by leveraging excess storage. In [9], the authors analyze the communication load of the coded shuffling algorithm assuming that the number of data points approaches infinity. They also provide simulation results based on the coded shuffling algorithm and the statistical gains of without-replacement sampling, showing that coded shuffling algorithm can help parallel/distributed gradient methods converge faster. However, the authors did not provide experimental results. UBERSHUFFLE is a strict improvement over the original coded shuffling algorithm. The details are provided in Sec. 3.

A few recent works also study the fundamental limits of the coded shuffling problem [23, 24]. In [23], the authors characterize the information-theoretic limits of the coded shuffling problem for $n = 2$ and $n = 3$, and in [25], the authors consider the worst-case shuffling performance, and propose a new coded shuffling algorithm, designed to perform well under the worst-case scenario.

We note that a similar idea that uses coding ideas to reduce the shuffling communication load is also proposed in [26]. More precisely, the authors consider the MapReduce framework, in which distributed compute nodes first perform 'map' phase, then shuffle the results of the map phase between the compute nodes, and then perform 'reduce' phase to finish the computation. The key idea is to assign duplicate map tasks across the compute nodes to generate redundant map results. Viewing these redundant computation results as side-information, they apply coding-theoretic techniques to reduce the bandwidth required to shuffle the map results between the compute nodes. Even though this algorithm seems very similar to ours, there are a few key differences. First, our algorithm is specifically designed to speed up the convergence of distributed machine learning algorithms while theirs is for MapReduce tasks. Further, our algorithm makes use of the excess memory/storage to generate common information while their algorithm generates common information by increasing the computational load of the map phase by a constant factor.

1.1.3 Efficient Multicast Algorithms

In [27], the authors propose a multicast algorithm that transmits data from a node to multiple other nodes in about twice of the unicast time. Using this algorithm, we analyze the shuffling time for coded shuffling algorithm when used without a perfect broadcast channel. In Sec. 3, we provide the details of the multicast algorithm and new guarantees of the coded shuffling algorithm when used without a broadcast channel.

2 System Model and Data Shuffling Algorithms

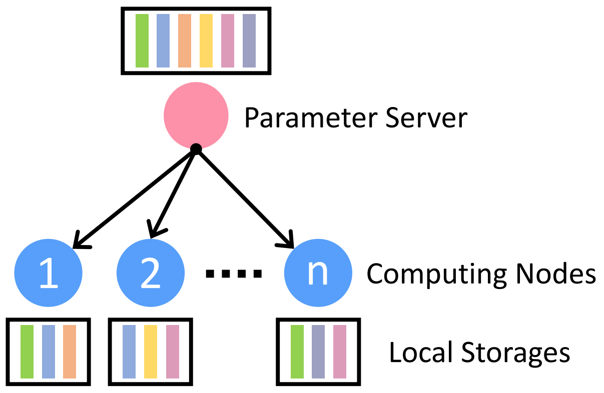

Figure 1: Illustration of the distributed setup. The parameter server has access to entire data points, and the compute nodes can store a limited number of data points.

In this section, we first describe the system model and then provide informal illustrations of the shuffling algorithms, proposed in [8], using a running example. See Fig. 1 for a toy illustration. Consider a distributed computing environment with n distributed compute nodes and a parameter server. The parameter server 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. The parameter server first draws a random assignment of each data point such that all compute nodes are assigned the same number of data points. Let us denote the destination of data point d_i by $w_i \in [n]$. We denote by D_i the set of data indices assigned to compute node i, i.e., $D_i = \{j \mid w_j = i\}$. We also assume that each compute node can cache up to $s := \lfloor \alpha q \rfloor$ data points, for $1/q \leq \alpha < 1$.

We denote the set of data indices that is cached in compute node i's cache by C_i. [1]

2.1 The Uncoded Shuffling Algorithm

At the beginning of epoch $t+1$, the parameter server first draws a new data assignment w^t, and then computes $U_i^{t+1} := D_i^{t+1} \setminus C_i^t$ for each i. Note that U_i^{t+1} is a set of data points that are required by compute node i for epoch $t+1$ but are not stored in compute node i after epoch t. Under the uncoded shuffling algorithm, the parameter server shuffles data points simply by transmitting U_i^{t+1} to compute node i for each i. It is clear that every compute node is able to obtain its own data set because $D_i^{t+1} = U_i^{t+1} \cup \left(C_i^t \cap D_i^{t+1}\right)$ for all i.

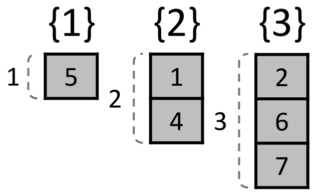

Figure 2: Illustration of the encoding table for uncoded shuffling algorithm.

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 compute node 1, d_1, d_4 to compute node 2, and d_2, d_6, d_7 to compute node 3. See Fig. 2 for visualization.

In general, the communication cost of the uncoded shuffling algorithm can be shown as following theorem [9].

Theorem 1 (Uncoded Shuffling Rate). *Total communication rate (in data points transmitted per epoch) of the uncoded shuffling scheme is*

$$R_u = q(1 - s/q). \tag{1}$$

Proof. Since the new permutation S_i^{t+1} is picked uniformly at random, s/q fraction of the data rows in S_i^{t+1} are present in current cache C_i^t. Hence, the number of data that each compute node need becomes $|C_i^t \cap S_i^{t+1}| = \frac{q}{n}(1 - \frac{s}{q})$. Thus, the parameter server needs to transmit $n \times \frac{q}{n}(1 - \frac{s}{q}) = q(1 - \frac{s}{q})$ data points in total. □

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 parameter server 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 compute node can obtain new data points by decoding the received message with the aid of cached data points.

The coded shuffling algorithm consists of 3 parts: indexing algorithm where encoding table is generated, encoding algorithm where data are encoded based on encoding table, and decoding algorithm where encoded data is decoded to obtain the data points required by each compute nodes.

Indexing Algorithm. At the beginning of epoch $t+1$, the parameter server first constructs a table that contains the information about which data points are cached by which compute nodes after epoch t. This information can be represented by a binary-valued matrix $\mathbf{C}^t \in \{0,1\}^{n \times m}$, whose (i,j)-th element represents whether data point j is cached at compute node i (1) or not (0). Consider data point j, which is assigned to compute node $\pi(j)$ for epoch $t+1$. The parameter server finds the set of compute nodes in which this data point is cached, i.e., $\{i : \mathbf{C}_{ij}^t = 1\}$,

[1]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 [28].) 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.

Algorithm 1 Coded Shuffling: Indexing algorithm at parameter server

```
procedure INDEXING($C, \pi$)                                  ▷ $\pi(j) = i$: row $j$ is assigned to compute node $i$
    $Z(\mathcal{I}, i) \leftarrow \emptyset$, $\forall I \in [n]^n, i \in \mathcal{I}$      ▷ indices for compute nodes in $\mathcal{I}$ and assigned to compute node $i$
    for each data point $a_j$ in $A$ do
        if $C_{\pi(j),j} = 1$ then                            ▷ $a_j$ is already cached at compute node $i$
            continue
        end if
        $\mathcal{I} \leftarrow \{i : C_{ij} = 1\} \cup \{\pi(j)\}$
        $Z(\mathcal{I}, \pi(j)) \leftarrow Z(\mathcal{I}, \pi(j)) \cup \{j\}$
    end for
    Return $Z$
end procedure
```

Algorithm 2 Coded Shuffling: Encoding algorithm at parameter server

```
procedure ENCODING($Z, A$)
    for each $\mathcal{I} \in [n]^n$, do
        $Y \leftarrow (Z(\mathcal{I}, \mathcal{I}_1), Z(\mathcal{I}, \mathcal{I}_2), \dots Z(\mathcal{I}, \mathcal{I}_{|\mathcal{I}|}))$      ▷ $\mathcal{I}_i$ is the $i^{\text{th}}$ element of $\mathcal{I}$ in (some order)
        $\ell \leftarrow \max_{i=1}^{|\mathcal{I}|} |Y_i|$
        for $t \in [1, 2, \dots, \ell]$ do
            broadcast $b_{\mathcal{I},t} = \sum_{i=1}^{\mathcal{I}} a_{(Y_i)_t}$ (with metadata $Y_t$)      ▷ if $|Y_i| < t$, $a_{(Y_i)_t} = 0$
        end for
    end for
end procedure
```

and assigns it to a tuple $(\mathcal{I}, \pi(j))$, where $\mathcal{I} := \{i : \mathbf{C}^t_{ij} = 1\} \cup \pi(j)$. We define $Z(\mathcal{I}, i)$ as the set of data points that are assigned to tuple $(\mathcal{I}, i)$. Note that all the data points in $Z(\mathcal{I}, i)$ are required by compute node i, and cached by the compute nodes in $\mathcal{I} \setminus \{i\}$.

Encoding Algorithm. After every data point is indexed in a similar way, the parameter server starts encoding the data points as follows. For each subset $\mathcal{I} = \{i_1, i_2, \dots, i_{|\mathcal{I}|}\}$ of $[n]$, it first looks up the following assignment sets: $Z(\mathcal{I}, i_1), Z(\mathcal{I}, i_2), \dots, Z(\mathcal{I}, i_{|\mathcal{I}|})$. All the data points in these assignment sets will be multicasted to compute nodes in $\mathcal{I}$. Given these sets, the parameter server removes one data point from each of the $|\mathcal{I}|$ assignment sets, and then generates an encoded data point by adding them up. This procedure is repeated until all the assignment sets become empty.

Decoding Algorithm. After multicasting is done, the compute nodes obtains the required data points by decoding the received packets based on the encoding table. For decoding, each compute node i identifies the encoded data points including D_i^{t+1}, and decodes the required data by subtracting the data rows cached in the compute node. Otherwise, the compute node simply discards the data received.

Note that cache information binary matrix $\mathbf{C}^t_{ij}$ and data assignment for next epoch π, generated by the parameter server, are broadcast to every compute nodes. After receiving this information, each compute node identifies assignment sets $Z(\mathcal{I}, i)$ of the encoding table by running the indexing algorithm, given in Algorithm 1.

The pseudocode of the indexing, encoding and decoding algorithms are provided in Alg. 1, Alg. 2 and Alg. 3. We also note that the computational overhead of the indexing and encoding procedures is $\mathcal{O}(nq)$, which is observed to be negligible compared to the time to communicate the training data in Sec. 4.

We now illusrate the indexing, encoding, and decoding procedures of the algorithm in following toy example.

Example 2 (The coded shuffling algorithm). See Fig. 3 for visualization. Consider the same scenario considered in the previous example shown in Example 1. Assume that the parameter server *broadcasts* $d_2 + d_5$. Since d_2 is stored in compute node 1 and d_5 is stored in compute node 3, compute node 1 can obtain d_5 by subtracting d_2 from $d_2 + d_5$, and compute node 3 can obtain d_5 similarly. Thus, the parameter server can reallocate two data points using a single broadcast transmission, reducing the communication overheads. The coded shuffling algorithm is a straightforward generalization of this idea, which we briefly explain below. Consider data point d_1, which is required by compute node 2 for the subsequent epoch of the learning algorithm. Observe that this data point is *exclusively* cached in compute node 3. On the other hand, the data points d_6 and d_7 are required by compute node 3, and they are cached in compute

Algorithm 3 Coded Shuffling: Decoding algorithm at compute node i

```
procedure DECODING($b, Y, \pi$)                          ▷ $i$ is the compute node index
    $J \leftarrow \{j : \pi(j) = i\} \cap Y$                     ▷ $|J|$ is always 0 or 1
    if $|J| = 0$ then
        return null
    else
        return $b - \sum_{i \in Y \setminus J} a_i$
    end if
end procedure
```

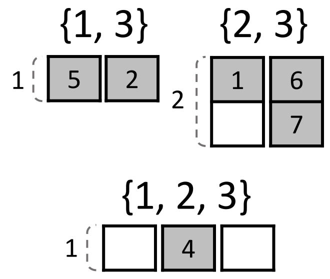

Figure 3: Illustration of the encoding table for coded shuffling algorithm.

node 2. Hence, the parameter server can combine the data points from each of these groups and broadcast the encoded data points. One can visualize this indexing result in the table labeled as $\{2, 3\}$ in Fig. 3. The first column of the table corresponds to the data points that are required by compute node 2 and exclusively cached in $\{2, 3\} \setminus \{2\}$, and the second column is for the data points required by compute node 3 and exclusively cached in $\{2, 3\} \setminus \{3\}$. By repeating this procedure for every data point, one can obtain all the encoding tables shown in Fig. 3. Once the encoding table is obtained, the parameter server simply generates one encoded packet per row of the encoding tables, and broadcasts them to the compute nodes. By the property of the indexing procedure, every compute node is guaranteed to obtain the desired data points by decoding the received packets. Here, note that the communication cost of the coded shuffling algorithm for this example is 4 transmissions.

The communication cost of the coded shuffling algorithm is characterized in [8].

Theorem 2 ([8] 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 $p = \frac{s-q/n}{q-q/n}$.*

Proof. The authors of [8] shows that cardinality of set $S_i^{t+1} \cap \tilde{C}^t_{\mathcal{I}\setminus\{i\}}$ for $\mathcal{I} \in [n]$ with $|\mathcal{I}| \geq 2$ is

$$|S_k^{t+1} \cap \tilde{C}^t_{\mathcal{I}\setminus\{i\}}| \simeq \frac{q}{n} \times \frac{|\mathcal{I}|-1}{n} p^{|\mathcal{I}|-2}(1-p)^{n-(|\mathcal{I}|-1)},$$

when q gets large (and n remains sub-linear in q).

Since the parameter server will send $\sum_{i\in\mathcal{I}} A(S_i^{t+1} \cap \tilde{C}^t_{\mathcal{I}\setminus\{i\}})$ for each subset $\mathcal{I}$ with $|\mathcal{I}| \geq 2$, the total rate of coded transmission is

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

To complete the proof, above expression can be simplified using the following identity:

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

□

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 parameter server and the compute nodes and denoting the shuffling time by T, we can characterize the shuffling times as follows.

Corollary 3 (Shuffling times with broadcasts). *Given the broadcast bandwidth B, 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 UberShuffle: The Coded Shuffling Algorithm with Carpool

The coded shuffling algorithm cannot be immediately deployed in practice since its theoretical guarantee holds only when the number of data points approaches infinity. That is, it is not clear whether or not the algorithm can achieve its promised performance under practical settings with a finite number of data points. In this section, we propose a modification to the coded shuffling algorithm to address this limitation, making the algorithm more applicable in practice.

In Thm. 2, it makes an implicit assumption that the number of data points q grows faster than a certain growth rate. Recall that the parameter server 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 $q = \omega(e^n)$.[2]

To improve the performance of the coded shuffling algorithm in practice, we propose a variation of it, which we call UberShuffle. We illustrate the new encoding algorithm via the running example.

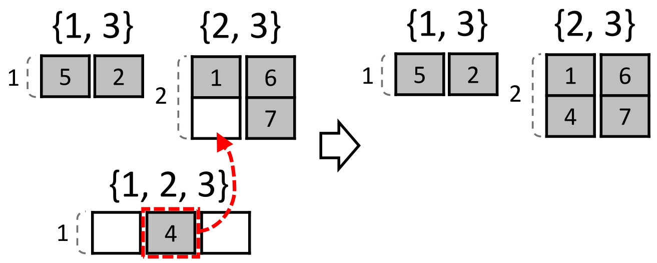

Figure 4: Illustration of the encoding table for UberShuffle algorithm.

Example 3 (UberShuffle). See Fig. 4 for visualization. Recall that in Fig. 3, 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 compute node 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, any data point can be reallocated from column i of table $\mathcal{I}_1$ to column i of table $\mathcal{I}_2$ if $\mathcal{I}_1 \supset \mathcal{I}_2$. 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, the UberShuffle algorithm greedily optimizes the number of encoded packets by considering each layer of the DAG.

[2]A similar phenomenon (though not identical) is also observed in the coded caching literature. (See [29].)

To explain the algorithm in detail, the algorithm first constructs $n-1$ arbitrarily ordered lists of index sets: $\mathcal{J}_2, \mathcal{J}_3, \dots, \mathcal{J}_n$, where $\mathcal{J}_i = \{\mathcal{I} \in [n]^n | |\mathcal{I}| = i\}$ for all $2 \le i \le n$. It then check each of the elements in $\mathcal{J}_2$ one by one. Consider an element of $\mathcal{J}_2$, say $\mathcal{I}$, and denote by $\mathcal{I}_1$ and $\mathcal{I}_2$ the elements of $\mathcal{I}$. If $|Z(\mathcal{I}, \mathcal{I}_1)| = |Z(\mathcal{I}, \mathcal{I}_2)|$, the packets destined for the subset $\mathcal{I}$ is maximally efficient, and hence the algorithm proceeds to the next element in $\mathcal{J}_2$. If $|Z(\mathcal{I}, \mathcal{I}_1)| \neq |Z(\mathcal{I}, \mathcal{I}_2)|$, this implies that the packets destined to the subset $\mathcal{I}$ has some rooms to include more packets. Without loss of generality, assume that $|Z(\mathcal{I}, \mathcal{I}_1)| + m = |Z(\mathcal{I}, \mathcal{I}_2)|$. That is, there are m more packets destined to node $\mathcal{I}_2$ than to node $\mathcal{I}_1$. For this case, the algorithm greedily finds packets that can be 'carpooled'. As mentioned above, one can safely reallocate packets from $Z(\mathcal{I}', \mathcal{I}_1)$ to $Z(\mathcal{I}, \mathcal{I}_1)$ if $\mathcal{I}' \supset \mathcal{I}$. Denote the set of (strict) supersets of set $\mathcal{I}$ by $\mathcal{I}^S$. Our algorithm searches for valid reallocations by iterating the elements of $\mathcal{I}^S \cap \mathcal{J}_2, \mathcal{I}^S \cap \mathcal{J}_3, \dots, \mathcal{I}^S \cap \mathcal{J}_{2+\nu}$. The algorithm stops the search process when it makes m reallocations, when it finishes iterating all the elements, or when it reaches the maximum depth ν. After the search process, the algorithm proceeds to the next elements in $\mathcal{J}_2$, and the same procedure is repeated until all elements of the set are examined. The algorithm then moves on to lists of larger index sets $\mathcal{J}_3$, $\mathcal{J}_4$, and so on. See Algorithm 4 for the pseudocode of our algorithm.

Algorithm 4 Coded Shuffling: UBERSHUFFLE algorithm at parameter server

```
procedure UBERSHUFFLE($Z, \nu$)
    for each $i \in [2, \dots, n]$, do
        for each $\mathcal{I} \in \mathcal{J}_i$, do                 ▷ $\mathcal{J}_i := \{\mathcal{I} \in [n]^n | |\mathcal{I}| = i\}$ for all $2 \le i \le n$
            $\ell \leftarrow \max_{j=1}^{|\mathcal{I}|} |Z(\mathcal{I}, \mathcal{I}_j)|$
            for each $\mathcal{I}_j \in \mathcal{I}$, do
                if $|Z(\mathcal{I}, \mathcal{I}_j)| < \ell$, then
                    $m \leftarrow \ell - |Z(\mathcal{I}, \mathcal{I}_j)|$
                    for each $k \in [i+1, i+2, \dots, i+\nu]$, do
                        for $\mathcal{I}' \in \mathcal{I}^S \cap \mathcal{J}_k$, do     ▷ $\mathcal{I}^S$ is set of (strict) supersets of set $\mathcal{I}$
                            if $|Z(\mathcal{I}', \mathcal{I}_j)| > 0$, then
                                $m' \leftarrow \min\{m, |Z(\mathcal{I}', \mathcal{I}_j)|\}$
                                reallocate $m'$ elements from $|Z(\mathcal{I}', \mathcal{I}_j)|$ to $|Z(\mathcal{I}, \mathcal{I}_j)|$
                                $m \leftarrow m - m'$
                            end if
                            if $m = 0$, then
                                break
                            end if
                        end for
                    end for
                end if
            end for
        end for
    end for
end procedure
```

To observe the gain of UBERSHUFFLE, we simulate the number of broadcast transmissions of various shuffling algorithms. Shown in Fig. 5 are simulation results with $n = 20, \nu = 2$, 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. 2), 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 $\alpha = 0.325$.

4 System Implementation

In this section, 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. The overall architecture of UBERSHUFFLE system is given in Fig. 6.

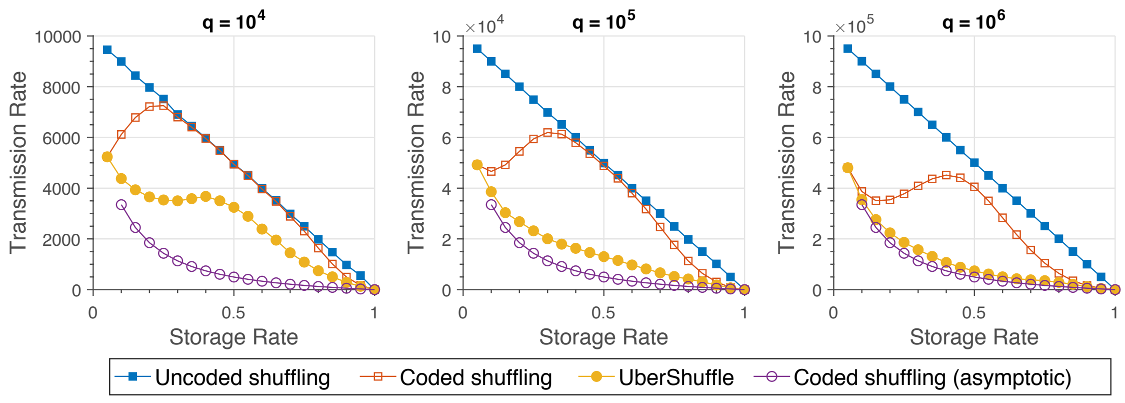

Figure 5: Number of packets used in Coded Shuffling algorithm for $n = 20$. Storage rate q/s is denoted by α.

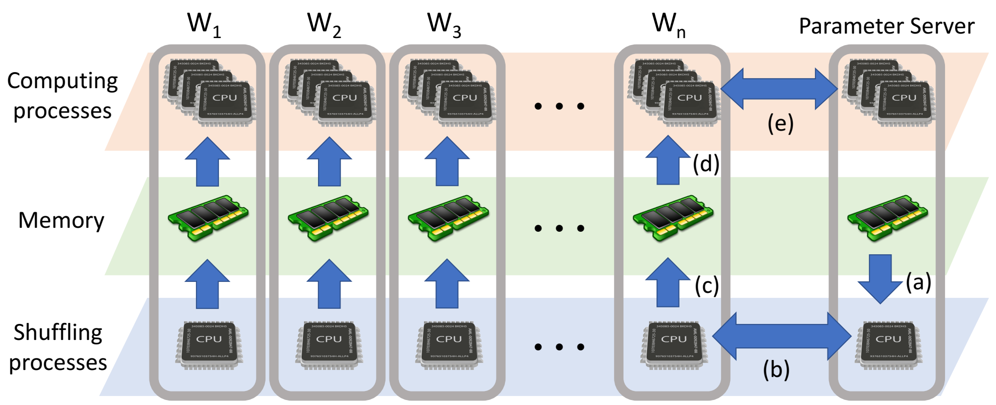

Figure 6: The system architecture of UBERSHUFFLE system. (The figure indicates a only some of nodes for simplicity.) Each of the stage indicates follows.(a) shuffling process of parameter server reads data, (b) and distributes shuffled data to shuffling process of compute nodes using shuffling algorithms. (c) Each compute node's shuffling processes saves shuffled data received from parameter server. (d) Each compute node's computing processes runs SGD algorithm using data saved by its shuffling process. (e) Models processed by computing process in each compute node are merged and distributed by parameter server's SGD process.

4.1 System Architecture

We design the system to use two cores in each node (including the parameter server), each of them is used for two different processes: computing process and shuffling process.

Initially, the entire training data is stored in parameter server's memory, and each of compute node stores a subset of parameter server's training data, with jointly covering the entire training data accross distributed nodes.

The computing processes in parameter server and compute nodes perform distributed SGD operations as a whole. On each epoch of the SGD procedure, each computing process runs SGD over existing local training data. After each epoch, the parameter server simply collects all the trained model from the n compute nodes, find a synchronized (averaged) model, and share the new model with the compute nodes.

The shuffling processes in parameter server and compute nodes shuffles the data across the nodes based shuffling algorithms explained in Sec. 2. On each epoch, The parameter server's shuffling core encodes the new data rows to be shuffled, and these encoded data rows are decoded by the compute nodes' shuffling cores. The compute nodes' shuffling cores are able to access the locally-cached data rows, which are necessary for decoding the encoded packets for coded shuffling.

We pipeline the epochs of SGD and shuffling procedures. That is, computing processes of all nodes runs SGD operation until newly shuffled data is ready in its local compute node's shuffling process. When shuffled data is ready, shuffling process copies its new data to computing process via memory in order for computing process to run SGD with newly received data. In this way, the system minimizes performance loss on SGD procedure due to waiting for shuffled data to be ready in each node.

4.2 Implementation of Efficient Multicasting Algorithms

In [9], the coded shuffling algorithm is shown to reduce the communication overhead by a factor of $\Theta(n)$ if the parameter server can transmit a packet to all n compute nodes in constant time. 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 another node at full bandwidth and at most one message can be received from another node at full bandwidth. Hence, a natural question is whether one can achieve the promised gain of the coded shuffling algorithm under this setup by multicasting a packet in constant time. It turns out that under a mild assumption, one can make use of an efficient multicast algorithm shown in the following theorem, similar to the one proposed in [27], which fully exploits interconnections between the compute nodes, to achieve the gain of the coded shuffling algorithm.

Theorem 4. *Assume a cluster with n compute nodes and a parameter server. There exists a multicast algorithm such that the time to multicast a single packet of unit size to n compute nodes is*

$$T_{\mathit{mutlicast}} = \frac{2n-1}{n} T_{\mathit{unicast}}. \tag{2}$$

Proof. Consider the following multicast algorithm. (See Fig. 7 for visual illustration.)

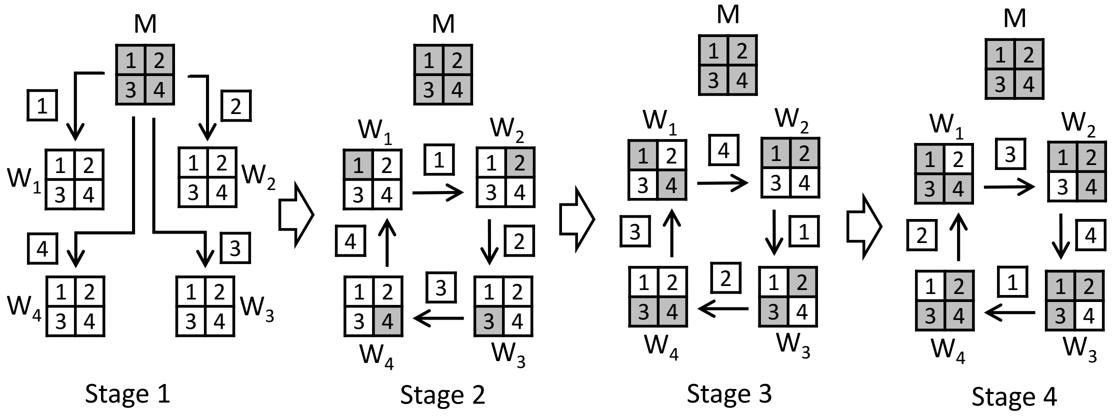

Figure 7: Illustration of our efficient multicast algorithm for $n = 4$. The packet is divided into n subpackets, indicated by square boxes. Dark boxes indicate that corresponding subpackets are already received.

In the first stage, the parameter server divides the packet into n subpackets and scatters them: the i^{th} packet is sent to compute node i for $1 \le i \le n$. In the subsequent stages, the n compute nodes circulate the subpackets, forming a ring structure. More precisely, for $1 \le j \le n-1$, in stage $j+1$, compute node i transmits $\{((i+j-2) \bmod n)+1\}^{\text{th}}$ subpacket to compute node $\{(i \bmod n)+1\}$. 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, the total transmission time is $\frac{2n-1}{n} T_{\text{unicast}}$. □

Applying the above theorem to our coded shuffling scheme, it is clear that for each packet, $1.5 T_{\text{unicast}} < T_{\text{mutlicast}} < 2 T_{\text{unicast}}$ since the number of destinations of each packet in our scheme is always larger than 1. Also, for large n, using Thm. 4, one can achieve $T_{\text{mutlicast}} \simeq 2 T_{\text{unicast}}$. 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.

Corollary 5 (Shuffling times with multicasts). *Given the unicast bandwidth B, the shuffling time of the coded shuffling algorithm with multicast channel is $T_{\mathit{multicast,coded}} = \frac{q}{B} \frac{(1-\alpha)}{\alpha n} \frac{2n-1}{n} \simeq \frac{q}{B} \frac{2(1-\alpha)}{\alpha n}$. Hence, $T_{\mathit{multicast,coded}} \simeq 2 T_{\mathit{broadcast,coded}} = \frac{2}{\alpha n} T_{\mathit{uncoded}}$.*

Unfortunately, the multicast algorithm proposed in Thm. 4 is not implemented in `OpenMPI` v1.6. Instead, we evaluate the performances of available multicast algorithms to decide which multicast algorithm to use in our system. Specifically, we compare the performance of the chain multicast algorithm, which is the closest one to our multicast algorithm, and that of the split-binary-tree algorithm [30]. For the evaluation, we run these multicast algorithms on an Amazon EC2 cluster with 20 instances. As a result, we observe that the split-binary-tree algorithm outperforms the other one in most scenarios: the packet tranmission time is upper bounded by $3.5T_{\text{unicast}}$ for all tested values of $n \leq 20$. [3]

4.3 Algorithm Modifications: Communicator Merging

Under `OpenMPI` framework, a *communicator* has to be defined for each set of multicasting group. Our coded shuffling algorithm considers all possible multicasting groups, which are 2^n combinations of n compute nodes. In OpenMPI v1.6, the large computational overhead is associated with communicator declaration. Hence, in the beginning of the algorithm, it is inevitable to declare all the communicators and keep them in memory. However, such an approach makes the coded shuffling algorithm infeasible due to the exponentially large memory footage. Moreover, in `OpenMPI` v1.6, only up to 2^{15} communicators can be declared at the same time.

In order to resolve this issue, we implement the communicator merging procedure, which merges 2^n communicators into $2^{n'}$ communicators, where n' is a chosen parameter which divides n. For instance, when $n = 20$ and $n' = 10$, the merging procedure reduces the number of communicators from $2^{20} \simeq 10^6$ to $2^{10} \simeq 10^3$, reducing the memory footage associated with communicators by a factor of 1000. We partition 20 compute nodes into 10 nodal subsets, with k^{th} subset containing compute node $2k-1$ and $2k$. When shuffling process of parameter server broadcasts encoded data to its target compute node processes, it concatenates data that has target node of same nodal subsets, and sends concatenated data based on 10 nodal subsets.

Define a function that maps an index set to an augmented index set as follows:

$$\psi_2(\mathcal{I}) = \{x | x \in [n], \left\lfloor \frac{x+1}{2} \right\rfloor = \left\lfloor \frac{i+1}{2} \right\rfloor \text{ for some } i \in \mathcal{I}\}. \tag{3}$$

With the above definition, one can see that the communication merging concatenates two encoded packets $b_{\mathcal{I}_1,t}$ and $b_{\mathcal{I}_2,t}$ if and only if $\psi_2(\mathcal{I}_1) = \psi_2(\mathcal{I}_2)$. For instance, $b_{\{2,3\},t}$ and $b_{\{1,3\},t}$ are concatenated since $\psi_2(\{2,3\}) = \psi_2(\{1,3\}) = \{1,2,3,4\}$.

We note that the communicator merging process slightly increases the shuffling time of coded shuffling since it increases the number of destinations per packet.

5 Experimental Results

In this section, we briefly introduce machine learning algorithms used in UBERSHUFFLE system. We then evaluate the performances of various shuffling algorithms on several applications.

5.1 Machine Larning Algorithms and Hardware Setup

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 [3].

The goal of the low-rank matrix completion problem is to fill the missing entries of a given matrix X of size $n_r \times n_c$ assuming that the rank of X is low compared to the dimension of it. This problem can be reduced to finding matrix L of size $n_r \times r$ and matrix R of size $n_c \times r$ by iteratively computing sequence of $L^{(k)}$ and $R^{(k)}$ as follows:

$$L_{i_k}^{(k+1)} = \left(\left(1 - \frac{\mu_1 \gamma_k}{|\Omega_{i_k\star}|}\right) L_{i_k}^{(k)} - \gamma_k f'_{i_k j_k} \left(L_{i_k}^{(k)} R_{j_k}^{(k)}\right) R_{j_k}^{(k)} \right), \tag{P5}$$

$$R_{j_k}^{(k+1)} = \left(\left(1 - \frac{\mu_1 \gamma_k}{|\Omega_{\star j_k}|}\right) R_{j_k}^{(k)} - \gamma_k f'_{i_k j_k} \left(L_{i_k}^{(k)} R_{j_k}^{(k)}\right) L_{i_k}^{(k)} \right), \tag{P6}$$

where Ω is a subset of $\{1,\dots,n_r\} \times \{1,\dots,n_c\}$, denoting the observed entries, and $\Omega_{i\star} = \{j | (i,j) \in \Omega\}$ and $\Omega_{\star j} = \{i | (i,j) \in \Omega\}$. The index pair (i_k, j_k) is chosen uniformly at random from Ω for k^{th} iteration, and γ_k is the step size for k^{th} iteration. Detailed proof of this property is given in [3].

In [3], the authors propose a way of distributing the observed entries of X across compute nodes and updating the parameters in parallel without incurring any conflicts. First, the parameter server process generates a random

[3]In [8], the authors experimentally show that the broadcasting time increases logarithmically in n while our results show that it is bounded by a constant. The difference is due to the different choices of multicasting algorithms: In [8], the multicasting algorithm is not specified, and hence rather inefficient algorithm is used by default; In this work, we specifically choose the split-binary-tree multicasting algorithm to achieve the constant multicasting time.

permutation of rows and columns, say π_{row} and π_{col}. It then partitions the entire data points into n^2 sets, where n is the number of compute nodes. Once the permutation is given, each compute node is assigned an exclusive set of n sets. The authors of [3] show that it is possible to compute the exact SGD updates in parallel even though n compute nodes work with their own individual data points in parallel. 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 more detailed description, see [3].

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 a fixed choice of number of rows n_r, number of column n_c, rank r, and noise level σ^2, we generate a low-rank matrix $M = Y_L Y_R^T$, where the entries of Y_L and Y_R are i.i.d. Gaussian random variables. We then normalize the generated matrices so that $\|M\| = 1$. Then, for each row of matrix M, a fixed number of entries are selected as training set (we denote the number of entries in each row by m). For the real data, we use the Movielens 20m dataset [31]. 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 $m = 68$.

We split the data set into training and test sets, and measure $(X_{ij} - M_{ij})^2 \,/\, (M_{ij})^2$ on the test set. We also measure the runtime of each algorithm.

On each instance, we run our system for 7 epochs, and the step size is reduced by a factor of 0.9 after every epoch, i.e., $\gamma_k = \gamma_0 (0.9)^k$. We evaluate the performances of the algorithms with varying r and μ, and here, for each configuration, we report the result with r and μ that perform the best only.

Parallel SGD for Linear Regression We also run the parallel SGD (PSGD) algorithm [1] for linear regression, described as follows. The parameter server randomly initialize x, and then broadcasts it to all compute nodes. At the same time, the parameter server reads the data matrix A and shuffles partial row vector a_i's of A and the corresponding y_i's to compute nodes. Once the data rows are shuffled, the compute nodes independently run the SGD algorithm using the local data rows. That is, for each data point a_i, the parameter server updates x as

$$x = x - \gamma a_i (a_i \cdot x - y_i), \tag{4}$$

where γ is the step size. After all the data points are used to update the parameter x, each compute node sends the final parameter x to the parameter server. The parameter server receives the updated parameters from the compute nodes, and then computes the global parameter x by averaging the received updated parameters. At the end of each epoch, we measure $\|x - x_{\text{ans}}\| / \|x_{\text{ans}}\|$ and its runtime. Synthetic data for our experiment is made by randomly generating a matrix A of size $q \times m$ and a vector x of size m, and y is generated by $y = Ax$.

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 parameter server 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 parameter server, and 20 `m3.xlarge` (4-core 2.5Ghz Intel Xeon E5-2670 v2 with 15GB RAM) instances for the compute nodes. Note that the parameter server 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 compute nodes are configured to have maximum network bandwidth, and to use only 2 cores as well.

5.2 Experimental Results

We summarize various experimental scenarios in Table 1 along with the measured shuffling times. In most cases, we observe that UBERSHUFFLE achieves the fastest shuffling times. Note that these shuffling times include all the extra overheads such as the encoding/decoding/indexing times. For the experimental setup (c), the uncoded shuffling time achieves the best performance: This is because the packet size for each data point ($m = 68$) is too small, and hence the communication costs of the shuffling algorithms are almost negligible. On the other hand, the coded shuffling algorithms cost non-negligible amounts of times to encode/decode/index data points, making themselves slower than the uncoded shuffling algorithm.

[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 [32].

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

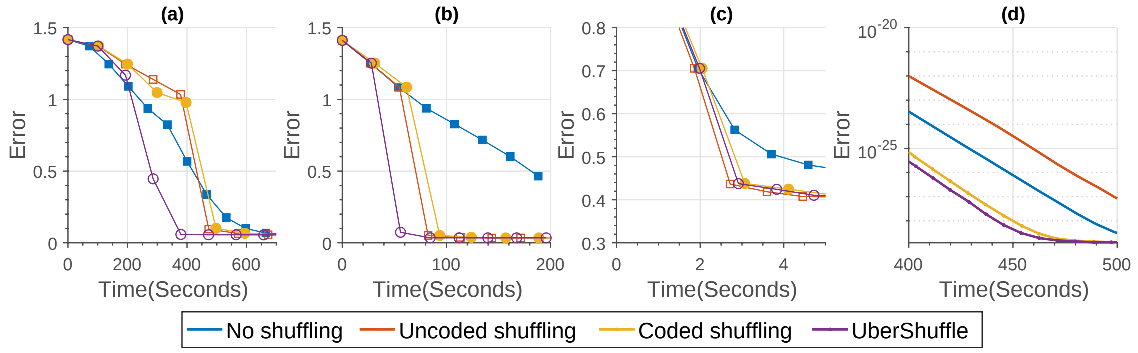

Figure 8: Convergence performances. (a) and (b) are for the matrix completion with synthetic data; (c) is for the matrix completion with real data; and (d) is for the linear regression with synthetic data.

We also report the convergence performance of the learning algorithms in Fig. 8. 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.

5.3 Data Shuffling via Shared Storage Systems

To deal with a large-scale data set in distributed machine learning systems, one may store the entire data set in a shared storage system, which can be accessed by the distributed compute nodes. If such a shared storage system is available, one can shuffle the data points simply reading data points from the shared storage system, without relying on the parameter server's shuffling mechanism. Compared with our data shuffling scheme, which fully exploits the network bandwidth among the distributed compute nodes, such an approach based on a shared storage system may incur various bottlenecks at the storage side, eventually slowing down the shuffilng process. In this section, we compare the performance of our shuffling systems with other distributed learning systems with a shared storage system.

We implement a new shuffling process that reads data points directly from a shared storage instead of receiving from the parameter server. At the beginning of each epoch, each shuffling process reads the data points that are required for the subsequent epoch but are not stored in the current cache, i.e., $S_i^{t+1} \setminus C_i^t$ for compute node i. Once the read process is completed, the new data points are copied to the SGD process of the same compute node. All the other experimental setups are kept the same as in Sec. 5.1.

For the shared storage system, we consider two options available on Amazon Web Service (AWS): EBS (Elastic Block Storage) and EFS (Elastic File System). For the EBS case, we attach a storage unit to the parameter server, and then let the parameter server share the disk with the distributed compute nodes using the NFS (Network File Storage) protocol. The EFS is a new virtual storage solution provided by AWS with which one can attach via NFS an networked storage system to compute units such as EC2 (Elastic Compute Cloud). There are two available operations modes: “General Purpose” mode and “Max I/O” mode. The “Max I/O” mode offers higher storage thoughput at the cost of higher file operation latency. On the other hand, “General Purpose” mode offers relatively low storage throughput but guarantees reasonable file operation latency.

We compare the performance of our systems with shuffling algorithms and those with shared storage systems. Especially, we run the SGD algorithm for low-rank matrix completion problem, described in Sec. 5.1, with $n_r = 300k, n_c = 300k, r = 10, m = 1000, \alpha = 0.05$. Table 2 summarizes the average per-epoch shuffling time.

Table 2: The average per-epoch shuffling time for UBERSHUFFLE and data shuffling by direct reading from shared storages.

Setup	Approx. Epoch Shuffling Time (sec)
UBERSHUFFLE	**34**
EBS Storage	110
EFS Storage (General purpose)	490
EFS Storage (Max I/O)	1100

One can clearly observe that the average per-epoch shuffling time is much larger when shared storage systems are used instead of our shufflng algorithm: the UBERSHUFFLE system is 3.2 times faster than the fastest storage-based alternative. As briefly mentioned above, this is due to the network and disk bottlenecks at the storage side and the large number of file operations incurred during the shuffling procedure. Especially, for the case of the EFS storage, each data file is split into multiple pieces and stored as multiple file units in order to maximize the throughput. This significantly degrades the perforamnce of the shuffling procedure for the following reason. In every epoch, each compute node sends its own 15000 file queries to the shared storage system, and this simultaneously happens at all the compute nodes. Hence, in total, about 3×10^5 files quires are received at the storage side, significantly slowing down the shuffling procedure.

References

[1] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[2] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. of the 25th Annual Conference on Neural Information Processing (NIPS)*, 2011, pp. 693–701.

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

[4] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade - Second Edition*, 2012, pp. 421–436. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35289-8_25

[5] C. Zhang and C. Re, "Dimmwitted: A study of main-memory statistical analytics," *PVLDB*, vol. 7, no. 12, pp. 1283–1294, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p1283-zhang.pdf

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

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

[8] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, 2017.

[9] ——, "Speeding up distributed machine learning using codes," in *the Workshop on ML Systems at NIPS*, 2015.

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

[11] A. Nedic and A. E. Ozdaglar, "Distributed subgradient methods for multi-agent optimization," *IEEE Transactions on Automatic Control*, vol. 54, no. 1, pp. 48–61, 2009. [Online]. Available: http://dx.doi.org/10.1109/TAC.2008.2009515

[12] S. P. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011. [Online]. Available: http://dx.doi.org/10.1561/2200000016

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

[14] J. C. Duchi, A. Agarwal, and M. J. Wainwright, "Dual averaging for distributed optimization: Convergence analysis and network scaling," *IEEE Transactions on Automatic Control*, vol. 57, no. 3, pp. 592–606, 2012. [Online]. Available: http://dx.doi.org/10.1109/TAC.2011.2161027

[15] J. Chen and A. H. Sayed, "Diffusion adaptation strategies for distributed optimization and learning over networks," *IEEE Transactions on Signal Processing*, vol. 60, no. 8, pp. 4289–4305, 2012. [Online]. Available: http://dx.doi.org/10.1109/TSP.2012.2198470

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proc. of the 26th Annual Conference on Neural Information Processing Systems (NIPS)*, 2012, pp. 1232–1240. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p716_yuchenglow_vldb2012.pdf

[18] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan, "MLbase: A distributed machine-learning system," in *Proc. of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013. [Online]. Available: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper118.pdf

[19] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: an API for distributed machine learning," in *Proc. of the IEEE 13th International Conference on Data Mining (ICDM)*, 2013, pp. 1187–1192. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2013.158

[20] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 583–598. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[21] B. Recht and C. Re, "Beneath the valley of the noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences," *ArXiv e-prints*, Feb. 2012.

[22] O. Shamir, "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., 2016, pp. 46–54. [Online]. Available: http://papers.nips.cc/paper/6245-without-replacement-sampling-for-stochastic-gradient-methods.pdf

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

[24] ——, "On the worst-case communication overhead for distributed data shuffling," in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sept 2016, pp. 961–968.

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

[26] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," in *Communication, Control, and Computing (Allerton), 2015 53rd Annual Allerton Conference on*. IEEE, 2015, pp. 964–971.

[27] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts, "Interprocessor collective communication library (intercom)," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*. IEEE, 1994, pp. 357–364.

[28] J.-B. Michel, Y. K. Shen, A. P. Aiden, A. Veres, M. K. Gray, T. G. B. Team, J. P. Pickett, D. Holberg, D. Clancy, P. Norvig, J. Orwant, S. Pinker, M. A. Nowak, and E. L. Aiden, "Quantitative analysis of culture using millions of digitized books," *Science*, 2010. [Online]. Available: http://www.sciencemag.org/content/331/6014/176.full

[29] 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*, vol. 62, no. 10, pp. 5524–5537, Oct 2016.

[30] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, Jun 2007. [Online]. Available: https://doi.org/10.1007/s10586-007-0012-0

[31] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 19:1–19:19, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2827872

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