Coded TeraSort

Songze Li*, Sucha Supittayapornpong*, Mohammad Ali Maddah-Ali†, and Salman Avestimehr*
*University of Southern California, †Nokia Bell Labs
Email: {songzeli,supittay}@usc.edu, mohammad.maddahali@nokia-bell-labs.com, avestimehr@ee.usc.edu

***Abstract*—We focus on sorting, which is the building block of many machine learning algorithms, and propose a novel distributed sorting algorithm, named `CodedTeraSort`, which substantially improves the execution time of the `TeraSort` benchmark in Hadoop MapReduce. The key idea of `CodedTeraSort` is to impose *structured* redundancy in data, in order to enable in-network coding opportunities that overcome the data shuffling bottleneck of `TeraSort`. We empirically evaluate the performance of `CodedTeraSort` algorithm on Amazon EC2 clusters, and demonstrate that it achieves $1.97\times$ - $3.39\times$ speedup, compared with `TeraSort`, for typical settings of interest.**

***Keywords*-Distributed Computing; Machine Learning; Sorting; MapReduce; Data Shuffling; Coding.**

I. Introduction

The modern paradigm for large-scale machine learning, scientific computing, and data analysis involves a massively large distributed system comprising individually small and relatively unreliable computing nodes made of commodity low-end hardware. Specifically, distributed systems like Apache Spark [1] and computational primitives like MapReduce [2], Dryad [3], and CIEL [4] have gained significant traction, as they enable the execution of production-scale tasks on data sizes of the order of tens of terabytes and more.

However, as we "scale out" computations across many distributed nodes, massive amounts of raw and (partially) computed data must be moved among nodes, often over many iterations of a running algorithm, to execute the computational tasks. This creates a substantial *communication bottleneck*. For example, by analyzing Hadoop traces from Facebook, it is demonstrated that, on average, 33% of the overall job execution time is spent on data shuffling [5]. This ratio can be much worse for sorting and other basis tasks underlying many machine learning applications. For example, as shown in [6], 50% - 70% of the execution time can be spent for data shuffling in applications including TeraSort, WordCount, RankedInvertedIndex, and SelfJoin.

Recently, it has been been shown that "coding" can provide novel opportunities to improve the run-time performance of machine learning applications. On one hand, coding can significantly slash the communication load of distributed computing by leveraging carefully designed redundant local computations at the nodes. In particular, a coding framework for MapReduce, named *Coded MapReduce*, has been proposed in [7]–[9], which assigns the computation of each Map task at r *carefully chosen* nodes (for some $r \in \mathbb{N}$), in order to enable in-network coding opportunities that reduce the communication load by $r\times$. For example, by redundantly computing each Map task at only *two* carefully chosen nodes, Coded MapReduce can reduce the communication load of MapReduce by 50%. On the other hand, it was shown in [10] that error correcting codes (e.g., Maximum-Distance-Separable codes) can be utilized to create redundant computation tasks for linear computations (e.g., matrix multiplication), which can effectively alleviate the straggler effect in many distributed machine learning algorithms. For example, as demonstrated in [10], coded computing reduces the average run-time of a gradient descent algorithm for linear regression by 31.3% to 35.7%.

The goal of this paper is to *demonstrate as a proof of concept the impact of coding in reducing the data shuffling load of distributed computing, and speeding up the overall computations*. We focus on "sorting", which is not only a basic benchmark for distributed computing systems like Hadoop MapReduce and Spark, but also a key step in many machine learning algorithms including recommender systems, SVD and many graph algorithms, and has data shuffling as its main bottleneck. Our main result is the development of a new distributed sorting algorithm, named `CodedTeraSort`, that imposes *structured* redundancy in data to enable coding opportunities for efficient data shuffling, which results in speeding up the state-of-the-art algorithms by $1.97\times$- $3.39\times$ in typical settings of interest.

To date, there have been many distributed sorting algorithms developed to perform efficient distributed sorting on commodity hardware (see, e.g., [11], [12]). Out of these algorithms, `TeraSort` [13], originally developed to sort terabytes of data [14], is a commonly used benchmark in Hadoop MapReduce [15]. In consistence with the general structure of a MapReduce execution, in a `TeraSort` execution, each server node first *maps* each data point it stores locally into a particular partition of the key space, then all the data points in the same partition are *shuffled* to a single node, on which they are sorted within the partition to *reduce* the final sorted output.

Out of the above three steps, the time spent in the Map and the Reduce phases of the computation can be reduced by paralleling onto more processing nodes, while the shuffle time will almost remain constant. This is because that no matter how large the cluster size is, almost as much as the entire raw dataset of data need to be transferred over the network. Hence, data shuffling often becomes the bottleneck of the performance of the `TeraSort` algorithm (see, e.g., [6], [16]). In this paper, we propose to leverage coding to overcome the shuffling bottleneck of `TeraSort`. In particular, we develop a novel distributed sorting algorithm, named `CodedTeraSort`, that incorporates the coding ideas in [9] to inject structured computation redundancy in Map phase of `TeraSort`, in order to cut down its shuffling load. At a high-level `CodedTeraSort` can be explained as the following:

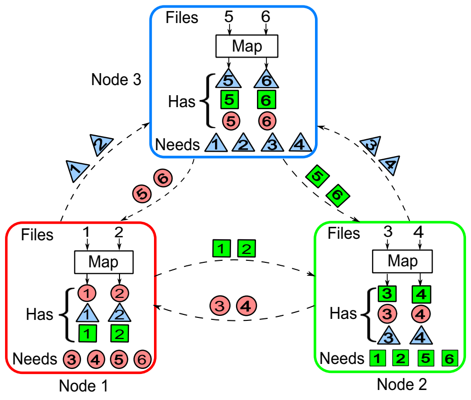
- The input data points are split into disjoint files, and each file is stored on *multiple carefully selected* server nodes to create *structured redundancy* in data.
- Each node maps all files that are assigned to it, following the Map procedure of `TeraSort`.
- Each node utilizes the imposed structured redundancy in data placement to create *coded packets* for data shuffling, such that the *multicast* of each coded packet delivers data points to several nodes simultaneously, hence speeding up the data shuffling phase.
- Each node decodes the data points that it needs for Reduce phase, from the received coded packets, and follows the Reduce procedure of `TeraSort`.

We empirically evaluate the performance of `CodedTeraSort` through extensive experiments over Amazon EC2 clusters. While the underlying EC2 networking does not support network-layer multicast, we perform the application-layer multicast for shuffling of coded packets, using the broadcast API `MPI_Bcast` from Open MPI [17]. Compared with the conventional `TeraSort` implementation, we demonstrate that `CodedTeraSort` achieves 1.97×-3.39× speedup for typical settings of interest. Despite the extra overhead imposed by coding (e.g., generation of the coding plan, data encoding and decoding) and application-layer multicasting, the practically achieved performance gain approximately matches the gain theoretically promised by the `CodedTeraSort` algorithm.

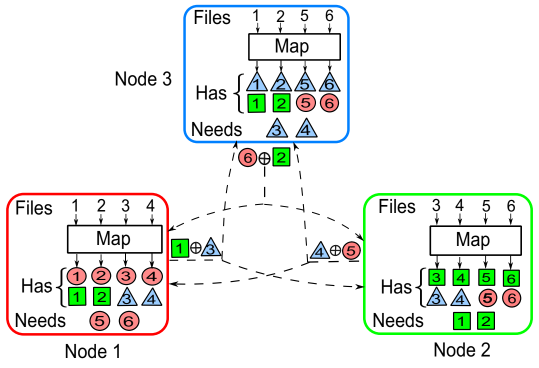
II. Overview of Coded MapReduce (CMR)

In this section, we provide an overview of the recently proposed *Coded MapReduce* (CMR) framework [7]–[9] via an example, to illustrate how coding can be utilized to reduce the data shuffling load in distributed computing.

Let us consider a general MapReduce-type framework for distributed computing, in which the overall computation is decomposed to three stages, *Map*, *Shuffle*, and *Reduce* that are executed distributively across many computing nodes. In the Map stage, each input file is processed locally, in one (or more) of the nodes, to generate *intermediate values*. In the Shuffle stage, for every output function to be calculated, all intermediate values corresponding to that function are transferred to one of the nodes for reduction. Finally, in the Reduce stage all intermediate values of a function are reduced to the final result.



(a) Uncoded MapReduce Scheme.



(b) Coded MapReduce Scheme.

Figure 1. Illustrations of the conventional uncoded MapReduce and the Coded MapReduce schemes to compute $Q = 3$ functions from $N = 6$ inputs using $K = 3$ nodes.

The driving idea for CMR is to leverage the available or under-utilized computing resources at various parts of the network, in order to create structured redundancy in computations that provides in-network coding opportunities for significantly reducing the data shuffling load. Let us illustrate CMR via a simple example.

Example (Coded MapReduce). Consider the MapReduce problem in Fig. 1 for distributed computing of 3 output functions, represented by red/circle, green/square, and blue/triangle respectively, from 6 input files, using three computing nodes. Nodes 1, 2, and 3 are respectively responsible for final reduction of red/circle, green/square, and blue/triangle output functions. Let us first consider the case where no redundancy is imposed on the computations, i.e., each file is mapped once. As shown in Fig. 1(a), Node i maps files $2i-1$ and $2i$ for $i = 1, 2, 3$. In this case, each node maps 2 input files locally[1], obtaining 2 out of 6 required intermediate values to reduce its output function. Hence, each node needs 4 intermediate values from the other nodes, yielding a communication load of $4 \times 3 = 12$.

[1]Note that when a node maps a file, it computes all three intermediate values of that file needed for the three output functions.

Now, we demonstrate how to leverage computation redundancy to slash the communication load via in-network coding. As shown in Fig. 1(b), computation load is doubled such that each file is now mapped on two nodes (files are cached prior to computations at the nodes). It is apparent that since more local computations are performed, each node now only requires 2 other intermediate values, and an uncoded shuffling scheme would achieve a communication load of $2 \times 3 = 6$. However, we can do much better with coding. As shown in Fig. 1(b), instead of unicasting individual intermediate values, every node multicasts XOR, denoted by $\oplus$, of 2 intermediate values to the other two nodes, simultaneously satisfying their data demands[2]. For example, knowing the blue/triangle in File 3, Node 2 can cancel it from the coded packet sent by Node 1, recovering the needed green/square in File 1. Therefore, this coding incurs a communication load of 3, achieving a $2\times$ gain from the uncoded shuffling. $\square$

More generally, we can consider a distributed computing scenario, where K nodes collaborate to compute Q arbitrary functions $(\phi_1, \ldots, \phi_Q)$ from N inputs $(x_1, \ldots, x_N)$ via a MapReduce-type framework, for $q = 1, \ldots, Q$:

$$\phi_q(x_1, \ldots, x_N) = \underbrace{g_q}_{\text{reduce}}\,(\underbrace{\ell_{q,1}}_{\text{map}}(x_1), \ldots, \underbrace{\ell_{q,N}}_{\text{map}}(x_N)). \quad (1)$$

For this scenario, the *computation load*, r, is defined as the average number of nodes that map each file (e.g., $r = 2$ in the example of Fig. 1(b)). Similarly, the *communication load*, L, is defined as the total amount of intermediate values (i.e., $\ell_{q,j}(x_j)'s$) needed to be exchanged across nodes in the data shuffling stage (normalized by QN), in order to compute all Q output functions.

In the general CMR scheme proposed in [7], [9], the computation of each Map task is repeated at r *carefully chosen* nodes (i.e., incurring computation load of r), in order to enable the nodes to exchange coded multicast messages that are *simultaneously* useful for r other nodes. As a result, CMR reduces the communication load by exactly a multiplicative factor of the computation load r (see Fig. 2), i.e., achieving

$$L_{\text{CMR}}(r) = \tfrac{1}{r} L_{\text{uncoded}}(r) = \tfrac{1}{r}(1 - \tfrac{r}{K}) = \Theta(\tfrac{1}{r}). \quad (2)$$

Also in [9], an information-theoretic lower bound on the minimum possible communication load, $L^*(r)$, was derived, which was demonstrated to exactly match that achieved by CMR, i.e., $L^*(r) = L_{\text{CMR}}(r) = \Theta(\frac{1}{r})$. Interestingly, this has revealed a fundamental *inversely-linear proportional* trade-off between computation load (r) and communication load (L), which can be utilized to optimally trade the available or under-utilized computing resources in the network for communication bandwidth.

[2]This type of coding was also utilized to solve the index coding problem [18], [19] that arises from the network coding problem [20].

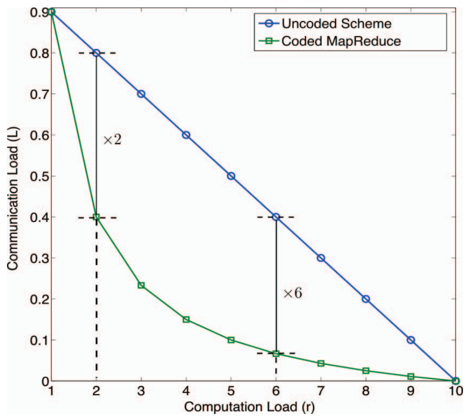


Figure 2. Comparison of the communication load of CMR with that of the uncoded schemes. For integer-valued computation load r, CMR achieves r times smaller communication load. Figure from [9].

CMR can also reduce the overall execution time by balancing the computation load in the Map stage and the communication load in the Shuffle stage. To illustrate this, let us consider a MapReduce application for which the overall response time is composed of the time spent executing the Map tasks, denoted by T_{map}, the time spent shuffling intermediate values, denoted by T_{shuffle}, and the time spent executing the Reduce tasks, denoted by T_{reduce}, i.e.,

$$T_{\text{total, MR}} = T_{\text{map}} + T_{\text{shuffle}} + T_{\text{reduce}}. \quad (3)$$

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

$$T_{\text{total, CMR}} \approx rT_{\text{map}} + \tfrac{1}{r}T_{\text{shuffle}} + T_{\text{reduce}}, \quad (4)$$

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

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

resulting in execution time of

$$T^*_{\text{total, CMR}} \approx 2\sqrt{T_{\text{shuffle}} T_{\text{map}}} + T_{\text{reduce}}. \quad (5)$$

For example, in a MapReduce application that T_{shuffle} is $10\times$ - $100\times$ larger than $T_{\text{map}} + T_{\text{reduce}}$, by comparing from (3) and (5), we note that CMR can reduce the execution time by approximately $1.5\times$ - $5\times$.

In the rest of this paper, we demonstrate how to utilize the ideas from CMR in order to develop a new distributed sorting algorithm, named `CodedTeraSort`, that leverages coding to speedup the conventional sorting algorithm, `TeraSort`. We will also empirically demonstrate the performance of `CodedTeraSort` via experiments over Amazon EC2 clusters.

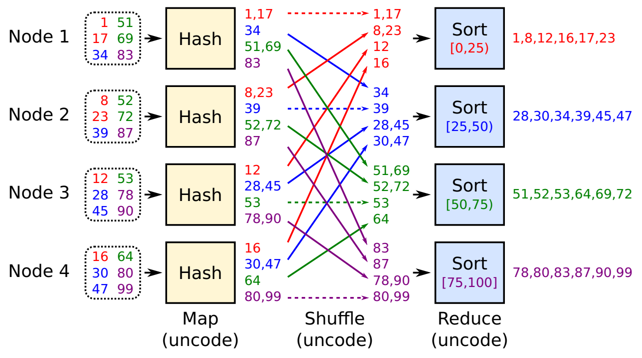


Figure 3. Illustration of conventional `TeraSort` with $K = 4$ nodes and key domain partitions $[0, 25), [25, 50), [50, 75), [75, 100]$. A dotted box represents an input file. An input file is hashed into 4 groups of KV pairs, one for each partition. For each of the 4 partitions, the 4 groups of KV pairs belonging to that partition computed on 4 nodes are all fetched to a corresponding node, which sorts all KV pairs in that partition locally.

III. TeraSort

`TeraSort` [14] is a conventional algorithm for distributed sorting of a large amount of data. The input data that is to be sorted is in the format of key-value (KV) pairs, meaning each input KV pair consists of a key and a value. For example, the domain of the keys can be 10-byte integers, and the domain of the values can be arbitrary strings. `TeraSort` aims to sort the input data according to their keys, e.g., sorting integers.

Let us consider `TeraSort` for distributed sorting over K nodes, whose indices are denoted by a set $\mathcal{K} = \{1, \dots, K\}$. The implementation consists of 5 components: File Placement, Key Domain Partitioning, Map Stage, Shuffle Stage, and Reduce Stage. In File Placement, the entire KV pairs are split into K disjoint files, and each file is placed on one of the K nodes. In Key Domain Partitioning, the domain of the keys is split into K partitions, and each node will be responsible for sorting the KV pairs whose keys fall into one of the partitions. In Map Stage, each node hashes each KV pair in its locally stored file into one of the K partitions, according to its key. In Shuffle Stage, the KV pairs in the same partition are delivered to the node that is responsible for sorting that partition. In Reduce Stage, each node locally sorts KV pairs belonging to its assigned partition. A simple example illustrating `TeraSort` is shown in Fig. 3. We next discuss each component in detail.

A. Algorithm Description

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

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

3) Map Stage: In this stage, each node hashes each KV pair in the locally stored file $F_{\{k\}}$ to the partition its key falls into. For each of the K key partitions, the hashing procedure on the file $F_{\{k\}}$ generates an *intermediate value* that contains the KV pairs in $F_{\{k\}}$ whose keys belong to that partition. More specifically, we denote the intermediate value of the partition P_j from the file $F_{\{k\}}$ as $I^j_{\{k\}}$, and the hashing procedure on the file $F_{\{k\}}$ is defined as

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

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

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

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

Since the partitions are created in the ascending order as specified in the above Key Domain Partitioning step, the collection of the K sorted list generated in the Reduce stage, i.e., $(Q_1, \dots, Q_K)$ represents the final sorted list of the entire input data.

B. Performance Evaluation

To understand the performance of `TeraSort`, we performed an experiment on Amazon EC2 to sort 12GB of data by running `TeraSort` on 16 nodes. The breakdown of the total execution time is shown in Table I.

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

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

We observe from Table I that for a conventional `TeraSort` execution, 98.4% of the total execution time was spent in data shuffling, which is 508.5× of the time spent in the Map stage. Given the fact that data shuffling dominates the job execution time, the principle of optimally trading computation for communication of Coded MapReduce reviewed in Section II can be applied to significantly improve the performance of `TeraSort`. For example, when executing the same sorting job using a coded version of `TeraSort` with a computation load of $r = 10$ (i.e., each input file is repeatedly mapped on 10 servers), we could

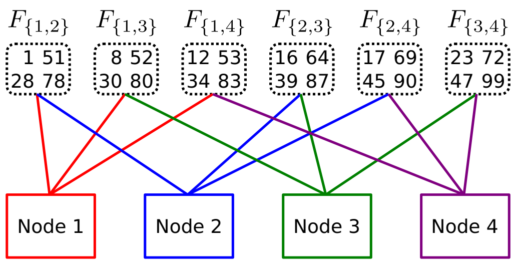


Figure 4. An illustration of the structured redundant file placement in `CodedTeraSort` with $K = 4$ nodes and $r = 2$.

theoretically save the total execution time by approximately $8\times$. This great promise of using CMR to improve the performance of `TeraSort` motivates us to develop a novel coded distributed sorting algorithm, named `CodedTeraSort`, which integrates the coding technique of CMR into the above described `TeraSort` algorithm to reduce the total execution time. We describe `CodedTeraSort` in detail in the next section.

IV. Coded TeraSort

In this section, we describe the `CodedTeraSort` algorithm, which is developed by integrating the coding techniques of the Coded MapReduce scheme illustrated in Section II into the above described `TeraSort` algorithm. `CodedTeraSort` exploits redundant computations on the input files in the Map stage, enabling in-network coding opportunities to significantly slash the load of data shuffling.

`CodedTeraSort` sorts a group of input KV pairs distributedly over K nodes, through the following 6 stages of operations:

1) *Structured Redundant File Placement*. The entire input KV pairs are split into many small files, each of which is repeatedly placed on multiple nodes according to a particular pattern.
2) *Map*. Each node applies the hashing operation as in `TeraSort` on each of its assigned files.
3) *Encoding to Create Coded Packets*. Each node generates coded multicast packets from local results computed in Map stage.
4) *Multicast Shuffling*. Each node multicasts each of its generated coded packet to a specific set of other nodes.
5) *Decoding*. Each node locally decodes the required KV pairs from the received coded packets.
6) *Reduce*. Each node locally sorts the KV pairs within its assigned partition.

Next, we describe the above 6 stages in detail.

A. Structured Redundant File Placement

For some parameter $r \in \{1, \dots, K\}$, we first split the entire input KV pairs into $N = \binom{K}{r}$ input files. Unlike the file placement of `TeraSort`, `CodedTeraSort` places each of the N input files *repetitively* on r distinct nodes.

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

$$\{F_{\mathcal{S}} : \mathcal{S} \subseteq \mathcal{K}, |\mathcal{S}| = r\}. \tag{6}$$

For example, when $K = 4$ and $r = 2$, the set of the input files is $\left\{F_{\{1,2\}}, F_{\{1,3\}}, F_{\{1,4\}}, F_{\{2,3\}}, F_{\{2,4\}}, F_{\{3,4\}}\right\}$.

We repetitively place an input file $F_{\mathcal{S}}$ on each of the r nodes in $\mathcal{S}$, and hence each node now stores $Nr/K = \binom{K-1}{r-1}$ files. As illustrated in a simple example in Fig. 4 for $K = 4$ and $r = 2$, the file $F_{\{2,3\}}$ is placed on Nodes 2 and 3. Node 2 has files $F_{\{1,2\}}, F_{\{2,3\}}, F_{\{2,4\}}$. We note that this redundant file placement strategy induces a *structured* distribution of the input files such that every subset of r nodes have a unique file in common.

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

B. Map

In this stage, each node repeatedly performs the Map stage operation of `TeraSort` described in Section III-A3, on each input file placed on that node. Specifically, for each file $F_{\mathcal{S}}$ with $k \in \mathcal{S}$ that is placed on Node k, Node k hashes the KV pairs in $F_{\mathcal{S}}$ to generate a set of K intermediate values $\left\{I^1_{\mathcal{S}}, \dots, I^K_{\mathcal{S}}\right\}$.

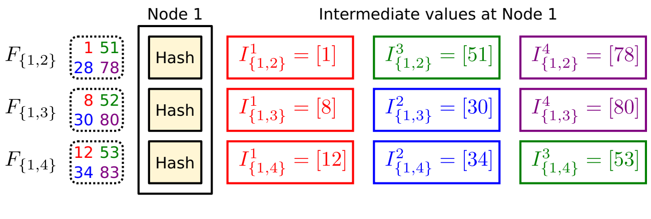


Figure 5. An illustration of the Map stage at Node 1 in `CodedTeraSort` with $K = 4$, $r = 2$ and the key partitions $[0, 25), [25, 50), [50, 75), [75, 100]$.

Only relevant intermediate values generated in the Map stage are kept locally for further processing. In particular, out of the K intermediate values $\left\{I^1_{\mathcal{S}}, \dots, I^K_{\mathcal{S}}\right\}$ generated from file $F_{\mathcal{S}}$, only $I^k_{\mathcal{S}}$ and $\left\{I^i_{\mathcal{S}} : i \in \mathcal{K} \backslash \mathcal{S}\right\}$ are kept at Node k. This is because that the intermediate value $I^i_{\mathcal{S}}$, required by Node $i \in \mathcal{S} \backslash \{k\}$ in the Reduce stage, is already available at Node i after the Map stage, so Node k does not need to keep them and send them to the nodes in $\mathcal{S} \backslash \{k\}$. For example, as shown in Fig. 5, Node 1 does not keep the intermediate value $I^2_{\{1,2\}}$ for Node 2. However, Node 1 keeps $I^1_{\{1,2\}}, I^3_{\{1,2\}}, I^4_{\{1,2\}}$, which are required by Nodes 1, 3, and 4 in the Reduce stage.

C. Encoding to Create Coded Packets

After the Map stage, each node has known locally a part of the KV pairs in the partition it is responsible for sorting, i.e., $\{I^k_{\mathcal{S}} : k \in \mathcal{S}, |\mathcal{S}| = r\}$ for Node k. In the next stages of

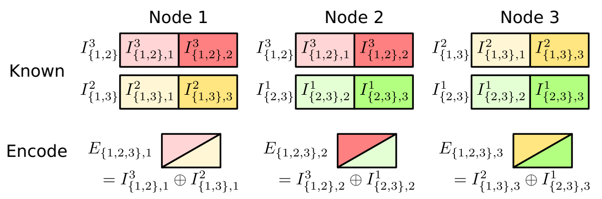

Node 1
Node 2
Node 3
Known
$I^3_{\{1,2\}}$ $I^3_{\{1,2\},1}$ $I^3_{\{1,2\},2}$
$I^2_{\{1,3\}}$ $I^2_{\{1,3\},1}$ $I^2_{\{1,3\},3}$
$I^3_{\{1,2\}}$ $I^3_{\{1,2\},1}$ $I^3_{\{1,2\},2}$
$I^1_{\{2,3\}}$ $I^1_{\{2,3\},2}$ $I^1_{\{2,3\},3}$
$I^2_{\{1,3\}}$ $I^2_{\{1,3\},1}$ $I^2_{\{1,3\},3}$
$I^1_{\{2,3\}}$ $I^1_{\{2,3\},2}$ $I^1_{\{2,3\},3}$
Encode
$E_{\{1,2,3\},1} = I^3_{\{1,2\},1} \oplus I^2_{\{1,3\},1}$
$E_{\{1,2,3\},2} = I^3_{\{1,2\},2} \oplus I^1_{\{2,3\},2}$
$E_{\{1,2,3\},3} = I^2_{\{1,3\},3} \oplus I^1_{\{2,3\},3}$

Figure 6. An illustration of the encoding process within a multicast group $\mathcal{M} = \{1, 2, 3\}$.

the computation, the server nodes need to communicate with each other to exchange the rest of the required intermediate values to perform local sorting in the Reduce stage.

The role of the encoding process is to exploit the structured data redundancy created by the particular repetitive file placement described above, in order to create coded multicast packets that are simultaneously useful for multiple nodes, thus saving the load of communicating intermediate values. For example in Fig. 5, Node 1 wants to send $I^2_{\{1,3\}} = [30]$ to Node 2 and $I^3_{\{1,2\}} = [51]$ to Node 3. Since $I^3_{\{1,2\}}$ and $I^2_{\{1,3\}}$ are already known at Node 2 and Node 3 respectively after the Map stage, instead of unicasting these two intermediate values individually, Node 1 can rather multicast a coded packet generated by XORing these two values, i.e., [30⊕51]. Then Node 2 and 3 can decode their required intermediate values using locally known intermediate values, e.g., Node 2 uses $I^3_{\{1,2\}} = [51]$ to decode $I^2_{\{1,3\}}$ by computing $I^2_{\{1,3\}} = [30 \oplus 51] \oplus [51] = [30]$. By multicasting a coded packet instead of unicasting two uncoded ones, we save the load of communication by 50%.

More generally, in the encoding stage, every node creates coded packets that are simultaneously useful for r other nodes. Specifically, in every subset $\mathcal{M} \subseteq \mathcal{K}$ of $|\mathcal{M}| = r+1$ nodes, the encoding operation proceeds as follows.

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

$$I^t_{\mathcal{M}\backslash\{t\}} = \{I^t_{\mathcal{M}\backslash\{t\},k} : k \in \mathcal{M}\backslash\{t\}\}, \tag{7}$$

  where $I^t_{\mathcal{M}\backslash\{t\},k}$ denotes the segment corresponding to Node k.
- For each $k \in \mathcal{M}$, we generate the coded packet of Node k in $\mathcal{M}$, denoted by $E_{\mathcal{M},k}$, by XORing all segments corresponding to Node k in $\mathcal{M}$, [3] i.e.,

$$E_{\mathcal{M},k} = \underset{t\in\mathcal{M}\backslash\{k\}}{\oplus} I^t_{\mathcal{M}\backslash\{t\},k}. \tag{8}$$

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

In Fig. 6, we consider a scenario with $r = 2$, and illustrate the encoding process in the subset $\mathcal{M} = \{1, 2, 3\}$. Exploiting the particular structure imposed in the stage of file placement, each node creates a coded packet that contains data segments useful for the other 2 nodes.

We summarize the the pseudocode of the Encoding stage at Node k in Algorithm 1.

```
// At Node $k$ //
// Data Segmentation
for each $\mathcal{M} \subseteq \mathcal{K}$ with $|\mathcal{M}| = r+1$ and $k \in \mathcal{M}$ do
  for each $t \in \mathcal{M}\backslash\{k\}$ do
    Consider file index $\mathcal{F} \leftarrow \mathcal{M}\backslash\{t\}$
    Evenly split $I^t_{\mathcal{F}}$ to $r$ segments $\{I^t_{\mathcal{F},j} : j \in \mathcal{F}\}$
  end for
end for
// Encode
for each $\mathcal{M} \subseteq \mathcal{K}$ with $|\mathcal{M}| = r+1$ and $k \in \mathcal{M}$ do
  Initialize coded packet $E_{\mathcal{M},k} \leftarrow \emptyset$
  for each $t \in \mathcal{M}\backslash\{k\}$ do
    Consider file index $\mathcal{F} \leftarrow \mathcal{M}\backslash\{t\}$
    $E_{\mathcal{M},k} \leftarrow E_{\mathcal{M},k} \oplus I^t_{\mathcal{F},k}$
    Store $\{I^t_{\mathcal{F},j} : j \in \mathcal{F}\backslash\{k\}\}$
  end for
  Store $E_{\mathcal{M},k}$
end for
```

Algorithm 1: Encoding to Create Coded Packets

D. *Multicast Shuffling*

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

As we have seen in the encoding process, each coded packet is simultaneously useful for r other nodes. Therefore, compared with an uncoded shuffling scheme that solely uses unicast communications, the multicast shuffling employed by `CodedTeraSort` reduces the communication load by exactly $r\times$. This gain is even higher compared with the `TeraSort` algorithm, for which no computation is repeated in the Map stage. This is because that even without multicasting, the redundant computations performed in the Map stage of `CodedTeraSort` already accumulate more locally available data needed for reduction, requiring less data to be shuffled across the network.

E. *Decoding*

During the stage of Multicast Shuffling, within each multicast group $\mathcal{M} \subseteq \mathcal{K}$ of $|\mathcal{M}| = r+1$ nodes, each Node $k \in \mathcal{M}$ receives a coded packet $E_{\mathcal{M},u}$ from Node u, for all $u \in \mathcal{M}\backslash\{k\}$. By the encoding process in (8), we have

$$E_{\mathcal{M},u} = \underset{t\in\mathcal{M}\backslash\{u\}}{\oplus} I^t_{\mathcal{M}\backslash\{t\},u}. \tag{9}$$

It is apparent that for all $t \in \mathcal{M}\backslash\{u, k\}$, we have $k \in \mathcal{M}\backslash\{t\}$, and Node k knows locally the intermediate

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

values $I^t_{\mathcal{M}\backslash\{t\}}$, for all $t \in \mathcal{M}\backslash\{u,k\}$, from the Map stage. Therefore, it knows locally all the data segments $\{I^t_{\mathcal{M}\backslash\{t\},u} : t \in \mathcal{M}\backslash\{u,k\}\}$. Then Node k performs the decoding process by XORing these data segments with $E_{\mathcal{M},u}$, i.e.,

$$E_{\mathcal{M},u} \oplus \left(\bigoplus_{t \in \mathcal{M}\backslash\{u,k\}} I^t_{\mathcal{M}\backslash\{t\},u} \right) = I^k_{\mathcal{M}\backslash\{k\},u}, \tag{10}$$

which recovers the data segment $I^k_{\mathcal{M}\backslash\{k\},u}$.

Similarly, Node k recovers all data segments $\{I^k_{\mathcal{M}\backslash\{k\},u} : u \in \mathcal{M}\backslash\{k\}\}$ from the received coded packets in $\mathcal{M}$, and merge them to obtain a required intermediate value $I^k_{\mathcal{M}\backslash\{k\}}$.

Finally, we repeat the above decoding process for all subsets of size $r+1$ that contain k, and Node k decodes the intermediate values $\{I^k_{\mathcal{M}\backslash\{k\}} : k \in \mathcal{M}, |\mathcal{M}| = r+1\}$, which can be equivalently represented by $\left\{I^k_{\mathcal{S}} : k \notin \mathcal{S}, |\mathcal{S}| = r\right\}$.

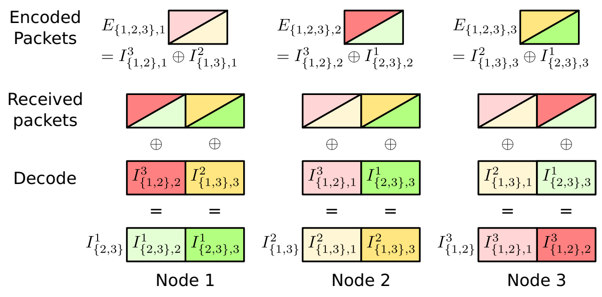

Encoded Packets
$E_{\{1,2,3\},1} = I^3_{\{1,2\},1} \oplus I^2_{\{1,3\},1}$
$E_{\{1,2,3\},2} = I^3_{\{1,2\},2} \oplus I^1_{\{2,3\},2}$
$E_{\{1,2,3\},3} = I^2_{\{1,3\},3} \oplus I^1_{\{2,3\},3}$
Received packets
Decode
$I^3_{\{1,2\},2}$ $I^2_{\{1,3\},3}$
$I^3_{\{1,2\},1}$ $I^1_{\{2,3\},3}$
$I^2_{\{1,3\},1}$ $I^1_{\{2,3\},3}$
$I^1_{\{2,3\}}$ $I^1_{\{2,3\},2}$ $I^1_{\{2,3\},3}$
$I^2_{\{1,3\}}$ $I^2_{\{1,3\},1}$ $I^2_{\{1,3\},3}$
$I^3_{\{1,2\}}$ $I^3_{\{1,2\},1}$ $I^3_{\{1,2\},2}$
Node 1
Node 2
Node 3

Figure 7. An illustration of the decoding process within a multicast group $\mathcal{M} = \{1,2,3\}$.

In Fig. 7, we consider a scenario with $r = 2$, and illustrate the above described decoding process in the subset $\mathcal{M} = \{1,2,3\}$. In this example, each node receives a multicast coded packet from each of the other two nodes. Each node decodes 2 data segments from the received coded packets, and merge them to recover a required intermediate value.

We summarize the the pseudocode of the Decoding stage at Node k in Algorithm 2.

```
// At Node $k$ //
for each $\mathcal{M} \subseteq \mathcal{K}$ with $|\mathcal{M}| = r+1$ and $k \in \mathcal{M}$ do
    Consider file index $\mathcal{F} \leftarrow \mathcal{M}\backslash\{k\}$
    for each $u \in \mathcal{F}$ do
        Initialize decoded segment $D^k_{\mathcal{F},u} \leftarrow E_{\mathcal{M},u}$
        for each Node $t \in \mathcal{F}\backslash\{u\}$ do
            Consider file index $\mathcal{W} \leftarrow \mathcal{M}\backslash\{t\}$
            $D^k_{\mathcal{F},u} \leftarrow D^k_{\mathcal{F},u} \oplus I^t_{\mathcal{W},u}$
        end for
    end for
    $I^k_{\mathcal{F}} \leftarrow$ Merge $\left\{D^k_{\mathcal{F},u} : u \in \mathcal{F}\right\}$
    Store $I^k_{\mathcal{F}}$
end for
```

Algorithm 2: Decoding

F. Reduce

After the Decoding stage, Node k has obtained all KV pairs in the partition P_k, for all $k \in \mathcal{K}$. In particular, the KV pairs $\left\{I^k_{\mathcal{S}} : k \in \mathcal{S}, |\mathcal{S}| = r\right\}$ are obtained locally in the Map stage, and the KV pairs $\left\{I^k_{\mathcal{S}} : k \notin \mathcal{S}, |\mathcal{S}| = r\right\}$ are obtained in the above Decoding stage.

In this final stage, Node k, $k = 1,\ldots,K$, performs the Reduce process as described in Section III-A5 for the `TeraSort`, sorting the KV pairs in partition P_k locally.

V. Evaluation

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

A. Implementation Choices

We first describe the following implementation choices that we have made for both `TeraSort` and `CodedTeraSort` algorithms.

Data Format: All input KV pairs are generated from `TeraGen` [13] in the standard Hadoop package. Each input KV pair consists of a 10-byte key and a 90-byte value. A key is a 10-byte unsigned integer, and the value is an arbitrary string of 90 bytes. The KV pairs are sorted based on their keys, using the standard integer ordering.

Platform and Library: We choose Amazon EC2 as the evaluation platform. We implement both `TeraSort` and `CodedTeraSort` algorithms in `C++`, and use Open MPI library [17] for communications among EC2 instances.

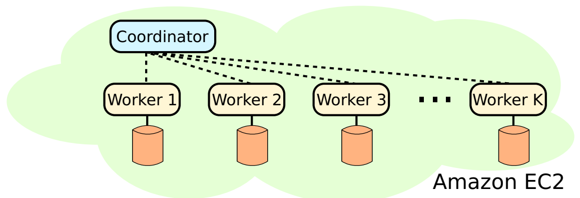


Figure 8. The coordinator-worker system architecture.

System Architecture: As shown in Fig. 8, we employ a system architecture that consists of a coordinator node and K worker nodes, for some $K \in \mathbb{N}$. Each node is run as an EC2 instance. The coordinator node is responsible for creating the key partitions and placing the input files on the local disks of the worker nodes. The worker nodes are responsible for distributedly executing the stages of the sorting algorithms.

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

In the TeraSort implementation, each node sequentially steps through Map, Pack, Shuffle, Unpack, and Reduce stages. The Map, Shuffle, and Reduce stages follow the descriptions in Section III. In the Reduce stage, the standard sort std::sort is used to sort each partition locally. To better interpret the experiment results, we add the Pack and the Unpack stages to separate the time of serialization and deserialization from the other stages. The Pack stage serializes each intermediate value to a continuous memory array to ensure that a single TCP flow is created for each intermediate value (which may contain multiple KV pairs) when MPI_Send is called[4]. The Unpack stage deserializes the received data to a list of KV pairs. In the Shuffle stage, intermediate values are unicast serially, meaning that there is only one sender node and one receiver node at any time instance. Specifically, as illustrated in Fig. 9(a), Node 1 starts to unicast to Nodes 2, 3, and 4 back-to-back. After Node 1 finishes, Node 2 unicasts back-to-back to Nodes 1, 3, and 4. This continues until Node 4 finishes.

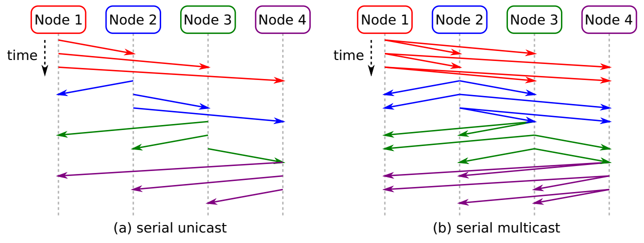


Figure 9. (a) Serial unicast in the Shuffle stage of TeraSort; a solid arrow represents a unicast. (b) Serial multicast in the Multicast Shuffle stage of CodedTeraSort; a group of solid arrows starting at the same node represents a multicast.

In the CodedTeraSort implementation, each node sequentially steps through CodeGen, Map, Encode, Multicast Shuffling, Decode, and Reduce stages. The Map, Encode, Multicast Shuffling, Decode, and Reduce stages follow the descriptions in Section IV. In the CodeGen (or code generation) stage, firstly, each node generates all file indices, as subsets of r nodes. Then each node uses MPI_Comm_split to initialize $\binom{K}{r+1}$ multicast groups each containing $r+1$ nodes on Open MPI, such that multicast communications will be performed within each of these groups. The serialization and deserialization are implemented respectively in the Encode and the Decode stages. In Multicast Shuffling, MPI_Bcast is called to multicast a coded packet in a serial manner, so only one node multicasts one of its encoded packets at any time instance. Specifically, as illustrated in Fig. 9(b), Node 1 multicasts to the other 2 nodes in each multicast group Node 1 is in. For example, Node 1 first multicasts to Node 2 and 3 in the multicast group $\{1,2,3\}$. After Node 1 finishes, Node 2 starts multicasting in the same manner. This process continues until Node 4 finishes.

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

B. Experiment Setup

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

- The coordinator runs on a r3.large instance with 2 processors, 15 GB memory, and 32 GB SSD.
- Each worker node runs on an m3.large instance with 2 processors, 7.5 GB memory, and 32 GB SSD.
- The incoming and outgoing traffic rates of each instance are limited to 100 Mbps.[5]
- 12 GB of input data (equivalently 120 M KV pairs) is sorted.

We evaluate the run-time performance of TeraSort and CodedTeraSort, for different combinations of the number of workers K and the parameter r. All experiments are repeated 5 times, and the average values are reported.

C. Experiment Results

The breakdowns of the execution times with $K = 16$ workers and $K = 20$ workers are shown in Tables II and III respectively. We observe an overall $1.97\times$-$3.39\times$ speedup of CodedTeraSort as compared with TeraSort. From the experiment results we make the following observations:

- For CodedTeraSort, the time spent in the CodeGen stage is proportional to $\binom{K}{r+1}$, which is the number of multicast groups.
- The Map time of CodedTeraSort is approximately r times higher than that of TeraSort. This is because that each node hashes r times more KV pairs than that in TeraSort. Specifically, the ratios of the CodedTeraSort's Map time to the TeraSort's Map time from Table II are $6.03/1.86 \approx 3.2$ and $10.84/1.86 \approx 5.8$, and from Table III are $4.68/1.47 \approx 3.2$ and $8.59/1.47 \approx 5.8$.
- While CodedTeraSort theoretically promises a factor of more than $r\times$ reduction in shuffling time, the actual gains observed in the experiments are slightly less than r. For example, for an experiment with $K = 16$ nodes and $r = 3$, as shown in Table II, the speedup of the Shuffle stage is $945.72/412.22 \approx 2.3 < 3$. This phenomenon is caused by the following two factors. 1) Open MPI's multicast API (MPI_Bcast) has an inherent overhead per a multicast group, for instance, a multicast tree is constructed before multicasting to a set of nodes. 2) Using the MPI_Bcast API, the time of multicasting a packet to r nodes is higher than that of unicasting the same packet to a single node. In fact,

[5]This is to alleviate the effects of the bursty behaviors of the transmission rates in the beginning of some TCP sessions. The rates are limited by traffic control command tc [21].

Table II
SORTING 12 GB DATA WITH $K = 16$ WORKER NODES AND 100 MBPS NETWORK SPEED

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

Table III
SORTING 12 GB DATA WITH $K = 20$ WORKER NODES AND 100 MBPS NETWORK SPEED

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

as measured in [10], the multicasting time increases logarithmically with r.

- The sorting times in the Reduce stage of both algorithms depend on the available memories of the nodes. `CodedTeraSort` inherently has a higher memory overhead, e.g., it requires persisting more intermediate values in the memories than `TeraSort` for coding purposes, hence its local sorting process takes slightly longer. This can be observed from the Reduce column in Tables II and III.
- The total execution time of `CodedTeraSort` improves over `TeraSort` whose communication time in the Shuffle stage dominates the computation times of the other stages.

Further, we observe the following trends from both tables:

The impact of redundancy parameter r: As r increases, the shuffling time reduces by approximately r times. However, the Map time increases linearly with r, and more importantly the CodeGen time increases as $\binom{K}{r+1}$. Hence, for small values of r ($r < 6$) we observe overall reduction in execution time, and the speedup increases. However, as we further increase r, the CodeGen time will dominate the execution time, and the speedup decreases. Hence, in our evaluations, we have limited r to be at most 5.[6]

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

In addition to the results in Tables II and III, we have performed more experiments, and listed their results on [22].

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

From those results, we observe that `CodedTeraSort` achieved up to $4.11\times$ speedup.

VI. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we integrate the principle of a recently proposed Coded MapReduce scheme into `TeraSort`, developing a novel distributed sorting algorithm `CodedTeraSort`. `CodedTeraSort` specifies a structured redundant placement of the input files that are to be sorted, such that the same file is repetitively processed at multiple nodes. The results of this redundant processing enable in-network coding opportunities that substantially reduce the load of data shuffling. We also empirically demonstrate the significant performance gain of `CodedTeraSort` over `TeraSort`, whose execution is limited by data shuffling.

Finally, we highlight three future directions of this work.

- *Beyond Sorting Algorithms.* Having successfully demonstrated the impact of coding in improving the performance of `TeraSort`, we can apply the coding concept to develop coded versions of many other distributed computing applications whose performance is limited by data shuffling (e.g., Grep, SelfJoin). In particular, mobile machine learning applications like mobile augmented reality and recommender systems are of special interest since the communications through wireless links are much slower. Also, we can study the connection between the CMR scheme and other types of coding in distributed computing (e.g., the Maximum-Distance-Separable codes used in [10]), in order to further optimize the run-time performance of general ditributed computing algorithms. Preliminary explorations along these directions have been made in [23]–[25].
- *Scalable Coding.* We observe from the experiment results that the coding complexity (i.e., the time spent at CodeGen stage) increases as $\binom{K}{r+1}$. Hence, as the redundancy parameter r gets large the coding overhead (including the time spent in generating the coding plan, encoding, and decoding) becomes comparable with or

even longer than the time spent in Map and Reduce stages. It is of great interest to design efficient and scalable coding procedures to maintain a low coding overhead.

- *Asynchronous Execution.* In the experiments, we executed the stages of the computation one after another in a synchronous manner. Also, the data shuffling was performed serially such that only one node is communicating (unicasting for `TeraSort` and multicasting for `CodedTeraSort`) at a time. It is interesting to explore the impact of coding in an asynchronous setting with parallel communications.

VII. Acknowledgement

This work is in part supported by NSF grants CAREER 1408639 and NETS-1419632, ONR award N000141612189, NSA award, and funds from Intel.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," *2nd USENIX HotCloud*, vol. 10, p. 10, June 2010.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Sixth USENIX OSDI*, Dec. 2004.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 59–72.

[4] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: a universal execution engine for distributed data-flow computing," in *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2011, pp. 113–126.

[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, Aug. 2011.

[6] Z. Zhang, L. Cherkasova, and B. T. Loo, "Performance modeling of MapReduce jobs in heterogeneous cloud environments," in *IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 839–846.

[7] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," *53rd Allerton Conference*, Sept. 2015.

[8] ——, "Fundamental tradeoff between computation and communication in distributed computing," *IEEE ISIT*, July 2016.

[9] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *e-print arXiv:1604.07086*, Apr. 2016, submitted to IEEE Trans. Inf. Theory.

[10] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *e-print arXiv:1512.02673*, Dec. 2015.

[11] S. G. Akl, *Parallel sorting algorithms*. Academic press, 2014, vol. 12.

[12] D. Pasetto and A. Akhriev, "A comparative study of parallel sort algorithms," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 203–204.

[13] "Hadoop TeraSort," https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html.

[14] O. OMalley, "Terabyte sort on Apache Hadoop," Yahoo, Tech. Rep., May 2008.

[15] "Apache Hadoop," http://hadoop.apache.org.

[16] Y. Guo, J. Rao, and X. Zhou, "iShuffle: Improving Hadoop performance with Shuffle-on-Write," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 107–117.

[17] "Open MPI: Open source high performance computing," https://www.open-mpi.org/.

[18] Z. Bar-Yossef, Y. Birk, T. Jayram, and T. Kol, "Index coding with side information," *IEEE Trans. Inf. Theory*, vol. 57, no. 3, Mar. 2011.

[19] S. El Rouayheb, A. Sprintson, and C. Georghiades, "On the index coding problem and its relation to network coding and matroid theory," *IEEE Trans. Inf. Theory*, vol. 56, no. 7, pp. 3187–3195, 2010.

[20] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Trans. Inf. Theory*, vol. 46, no. 4, July 2000.

[21] "tc - show / manipulate traffic control settings," http://lartc.org/manpages/tc.txt.

[22] "`CodedTeraSort` Implementations," *available online at: http://www-bcf.usc.edu/~avestime/CodedTerasort.html*, 2016.

[23] S. Li, Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "A scalable framework for wireless distributed computing," *e-print arXiv:1608.05743*, Aug. 2016, submitted to IEEE/ACM Trans. Netw.

[24] ——, "Edge-facilitated wireless distributed computing," in *Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM)*, Dec. 2016, pp. 1–7.

[25] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *Proceedings of the 2016 IEEE Workshop on Network Coding and Applications (NetCod)*, Dec. 2016.