# Leveraging Coding Techniques for Speeding up Distributed Computing

Konstantinos Konstantinidis and Aditya Ramamoorthy
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50010
Email: {kostas, adityar}@iastate.edu

***Abstract*—Large scale clusters running MapReduce, Spark etc. routinely process data that are on the orders of petabytes or more. The philosophy in these methods is to split the overall job into smaller tasks that are executed on different servers; this is called the map phase. This is followed by a data shuffling phase where appropriate data is exchanged between the servers. The final reduce phase, completes the computation.**

**Prior work has explored a mechanism for reducing the overall execution time by operating on a computation vs. communication tradeoff. Specifically, the idea is to run redundant copies of map tasks that are placed on judiciously chosen servers. The shuffle phase exploits the location of the nodes and utilizes coded transmission. The main drawback of this approach is that it requires the original job to be split into a number of map tasks that grows exponentially in the system parameters. This is problematic, as we demonstrate that splitting jobs too finely can in fact adversely affect the overall execution time.**

**In this work we show that one can simultaneously obtain low communication loads while ensuring that jobs do not need to be split too finely. Our approach uncovers a deep relationship between this problem and a class of combinatorial structures called resolvable designs. We present experimental results obtained on Amazon EC2 clusters for a widely known distributed algorithm, namely TeraSort. We obtain over 4.69$\times$ improvement in speedup over the baseline approach and more than 2.6$\times$ over current state of the art.**

## I. Introduction

In recent years, there has been a surge in the usage of various cluster computing frameworks such as MapReduce [1], Hadoop [2] and Spark [3]. The era of bigdata analytics [4] whereby a large amount of data needs to be processed in a fast manner has fueled this growth. In these applications, datasets are often so large that they cannot be housed in the memory and/or the disk of any one computer. Thus, the data is typically distributed across a number of nodes. Each node performs its own local computation, following which there is a shuffle phase where the nodes communicate among themselves. At this point the nodes perform the final computation. Henceforth, we refer to this as the MapReduce framework.

The MapReduce framework has proven to be quite versatile and large scale clusters in industry and academia routinely process terabytes of data using this approach. It is important to note that the framework intertwines computation and communication. Specifically, multiple servers allow for parallel computation; yet data needs to be exchanged between them to complete the processing of the job. It is well-recognized that the data shuffling phase that occurs between the map and the reduce steps, results in a significant amount of data movement. In fact, the data shuffling phase limits the performance of several applications [5]. Reference [6] suggests that for distributed machine learning algorithms the communication time can take up over $50\%$ of the overall job completion time.

The work of [7], [8] introduced a technique that allows a MapReduce-like system to operate on a tradeoff between communication and computation. For instance, consider a job being processed on very large file. The first step typically is to subdivide this file in smaller subfiles that can be processed on individual servers. The idea of [7], [8] is to process the same subfile at $r > 1$ carefully chosen servers. The choice of the servers allows for the usage of coded transmissions in the shuffle phase that can substantially reduce the induced communication load. For a given network throughput rate, a lower communication load translates into lesser time taken in the shuffle phase. Thus, the overall execution time of a given job can be reduced if the increased map phase execution time can be offset by the reduction in the shuffling time. The ideas in these works have their origins in the problem of coded caching [9].

### A. Main Contributions of our work

While the approach of [7], [8] is promising, there are some factors that limit the performance of their scheme. The approach of [10], [7], [8] crucially relies on subdividing the original file into a large number of subfiles. Henceforth we refer to the number of subfiles as the subpacketization level. In this work, we demonstrate that high subpacketization levels can significantly degrade the performance of the system. We propose alternate mechanisms that allow us to seamlessly tradeoff computation vs. communication, but with acceptable levels of subpacketization. Our mechanisms are based on a natural link between error correcting codes, combinatorial objects known as resolvable designs and MapReduce-like protocols (related links in a different context were explored in [11]). We note here that combinatorial designs have recently been explored to address issues in distributed storage systems [12], function computation over networks [13] and coded caching [11], [14]. We present exhaustive experimental comparisons with prior work that demonstrate the efficacy of our method.

This paper is organized as follows. Section II addresses our problem formulation. The specification of our protocol and its corresponding analysis appear in Section III. Details of

This work was supported in part by the National Science Foundation by grants CCF-1320416 and CCF-1718470.

978-1-5386-4727-1/18/$31.00 ©2018 IEEE

our implementation on Amazon EC2 clusters can be found in Section IV. Section V discusses our experimental results and several implementation related issues. Section VI concludes the paper. Owing to space limitations, certain proofs and in-depth discussions of results have been omitted from this paper. These can be found at [15].

## II. Problem Formulation

We now discuss the problem formulation more formally. Our presentation here is based closely on [7]. There are $N$ input subfiles that correspond to disjoint parts of the entire file to be processed. There are $Q$ arbitrary output functions that need to be computed across these $N$ subfiles. There are a total of $K$ homogeneous servers, i.e., machines that have similar computational power. For instance, in a word counting example, the subfiles could be the chapters of a book and the output functions are the word counts of a specific set of words. The subfiles will be denoted by $w_1, \ldots, w_N$ and the output functions $\phi_j, j = 1, \ldots, Q$. Each function $\phi_j$ depends on all the subfiles $w_1, \ldots, w_N$. We assume that the $j$-th function can be computed by a map phase followed by a reduce phase, i.e.,

$$\phi_j(w_1, \ldots, w_N) = h_j(g_{j,1}(w_1), \ldots, g_{j,N}(w_N)).$$

Here, $\mathbf{g}_n = (g_{1,n}, \ldots, g_{Q,n})$ "maps" the subfile $w_n$ into $Q$ intermediate values $\nu_{j,n}, j = 1, \ldots, Q$ each of which is assumed to be of size $B$ bits. The function $h_j$ maps the intermediate values $\nu_{j,n}$ on all subfiles into a reduced value $h_j(g_{j,1}(w_1), \ldots, g_{j,N}(w_N))$.

**Example 1.** Suppose that we consider the problem of counting $Q = 4$ words of a collection $\mathcal{A} = \{\text{and, if, when, the}\}$ in a book consisting of $N = 4$ chapters in a cluster with $K = 4$ servers. In this case the subfiles $w_1, \ldots, w_4$ are the chapters and $\phi_i, i = 1, \ldots, Q$ correspond to the counts of the words in $\mathcal{A}$ in the entire book, e.g., $\phi_1(w_1, \ldots, w_4)$ would be the number of occurrences of the word "and" in the book. Suppose that we define $\mathbf{g}_n$ to be the function that returns the counts of all the words in $\mathcal{A}$ in chapter $w_n$. Let us assume that the $i$-th slave is assigned subfile $w_i$ for all values of $i$. In this case it is evident that in the map phase, server $i$ computes $\mathbf{g}_i$ on its assigned subfile $w_i$ for $i = 1, \ldots, 4$. In the reduce phase, each server is given the responsibility of finding the overall count in the book of one specific word, e.g., suppose that server 1 reduces the word "and". In this case, it is evident that $\phi_1(w_1, \ldots, w_N)$ can be computed as

$$\phi_1(w_1, \ldots, w_N) = h_1(g_{1,1}(w_1), \ldots, g_{1,N}(w_N)).$$

In particular, the function $h_1$ simply corresponds to the sum of the counts of "and" on the individual chapters.

We define $r$ as the number of servers that map each subfile and for the remainder of the paper, we refer to it as the computation load.

**Definition 1.** The communication load $L \in [0, 1]$ of a certain scheme is defined as the ratio of the total number of bits transmitted in the data shuffling phase to $QNB$.

In Example 1, at the end of the map phase, each node needs three values from the other nodes. Thus, the total number of bits transmitted would be $4 \times 3 \times B = 12B$. Thus, the communication load of the system will be $L = 12B/16B = 3/4$.

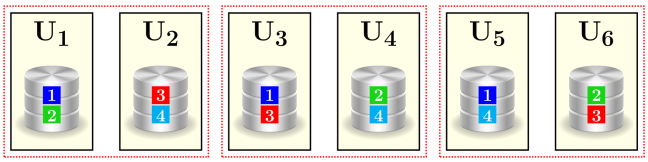


Fig. 1. Proposed placement scheme for $K = 6$ servers and $N = 4$ subfiles, denoted $\{1, 2, 3, 4\}$ and represented by colored squares, each assigned to some server. The red dashed boxes show the partition of the files into parallel classes.

We now present an example which demonstrates that increasing $r$ can translate into lower communication loads.

**Example 2.** Consider a system with $K = 6$ servers, a computation load of $r = 3$ and $Q = 6$ functions (e.g., word counts) that need to be computed. In our approach we would subdivide the original job into $N = 4$ subfiles (more generally any multiple of 4 can be used) that will be assigned to the servers as demonstrated in Fig. 1. At the end of the map step, each server would have computed the $Q$ functions on its assigned map tasks. Suppose that the $i$-th server is responsible for reducing the $i$-th function. This would imply, for example, that server $U_1$ needs the first function's evaluation on subfiles $w_3$ and $w_4$.

The key idea of our approach is for each server to transmit a packet that is simultaneously useful to multiple servers. For example, let us consider the group of servers $G_1 = \{U_1, U_3, U_6\}$ that were assigned subfiles $\{w_1, w_2\}, \{w_1, w_3\}$ and $\{w_2, w_3\}$, respectively. Then it is evident that at the end of the map phase, server $U_1$ wants $\nu_{1,3}$, server $U_3$ wants $\nu_{3,2}$ and server $U_6$ wants $\nu_{6,1}$. Let the intermediate value $\nu_{j,n}$ be encapsulated into a packet of size $B$ bits, denoted by $p(\nu_{j,n})$. Furthermore, consider subdividing this packet into two parts $p(\nu_{j,n})[1]$ and $p(\nu_{j,n})[2]$ (with size $B/2$ bits).

Now consider the set of transmissions specified in Table I. Note that server $U_1$ contains subfiles $w_1$ and $w_2$ and can therefore compute all $Q$ functions associated with them. Thus, it can transmit $p(\nu_{3,2})[1] \oplus p(\nu_{6,1})[2]$ as specified in row 1 of the table. Furthermore, it can be observed that this transmission is *simultaneously* useful to both servers $U_3$ and $U_6$. In particular, server $U_3$ already knows $p(\nu_{6,1})[2]$ and can therefore decode $p(\nu_{3,2})[1]$ which it wants. Likewise, server $U_6$ already knows $p(\nu_{3,2})[1]$ and can decode $p(\nu_{6,1})[2]$ that it wants. In a similar manner, it can be verified that each of the transmissions in Table I benefit two servers of the corresponding group. It turns out that the process of picking the servers to consider together can be made systematic; in addition to server group $G_1$ that we just considered, we can pick three others: $G_2 = \{U_1, U_4, U_5\}$, $G_3 = \{U_2, U_3, U_5\}$ and $G_4 = \{U_2, U_4, U_6\}$ which will result in all the servers obtaining their desired values.

The total number of bits transmitted in this case is therefore $4 \times 3 \times B/2 = 6B$; thus, the corresponding communication load is $\frac{6B}{QNB} = 0.25$. In contrast, uncoded transmission from the different nodes would have required a total of $2 \times 6 \times B = 12B$ bits to be transmitted, corresponding to a communication load of 0.5. Thus, the proposed approach reduces the communica-

TABLE I
CODED TRANSMISSIONS WITHIN ALL GROUPS IN EXAMPLE 2

| Group | Server | Transmission |
|---|---|---|
| $G_1$ | $U_1$ | $p(v_{3,2})[1] \oplus p(v_{6,1})[2]$ |
| | $U_3$ | $p(v_{6,1})[1] \oplus p(v_{1,3})[2]$ |
| | $U_6$ | $p(v_{3,2})[2] \oplus p(v_{1,3})[1]$ |
| $G_2$ | $U_1$ | $p(v_{5,2})[1] \oplus p(v_{4,1})[1]$ |
| | $U_4$ | $p(v_{5,2})[2] \oplus p(v_{1,4})[1]$ |
| | $U_5$ | $p(v_{4,1})[2] \oplus p(v_{1,4})[2]$ |
| $G_3$ | $U_2$ | $p(v_{5,3})[1] \oplus p(v_{3,4})[1]$ |
| | $U_3$ | $p(v_{5,3})[2] \oplus p(v_{1,1})[1]$ |
| | $U_5$ | $p(v_{3,4})[2] \oplus p(v_{1,1})[2]$ |
| $G_4$ | $U_2$ | $p(v_{6,4})[1] \oplus p(v_{4,3})[1]$ |
| | $U_4$ | $p(v_{6,4})[2] \oplus p(v_{2,2})[1]$ |
| | $U_6$ | $p(v_{2,2})[2] \oplus p(v_{4,3})[2]$ |

tion load in the shuffle phase by half.

We note here that the original work of [7] promises a communication load of $L_{coded}(r) = \frac{1}{r}(1-\frac{r}{K})$ for $r \in \{1, \ldots, K\}$. However, crucially this result assumes that

$$N = \binom{K}{r}\eta_1, \text{ where } \eta_1 \text{ is a positive integer.}$$

It is evident that $N$ grows very rapidly for their scheme. In Section V, we demonstrate that in real-life experiments the idealized analysis does not hold and the execution time suffers as a result of this.

For instance, for Example 2 above, their communication load would be $1/6$ (which is lower). However, their approach would require the original file to be split into $\binom{6}{3} = 20$ subfiles, i.e., their scheme only works when $N = 20$. In contrast, the scheme proposed in Example 2 works with $N = 4$ which is much smaller. In this work, we present significant generalizations of the basic approach in Example 2.

## III. IMPROVED SCHEMES FOR CODED DISTRIBUTED COMPUTATION FROM RESOLVABLE DESIGNS

We begin with some notions from combinatorial design theory [16].

**Definition 2.** A *design* is a pair $(X,\mathcal{A})$ consisting of

1) a set of elements (*points*), $X$, and
2) a family $\mathcal{A}$ (i.e. multiset) of nonempty subsets of $X$ called *blocks*, where each block has the same cardinality.

Thus, a design is simply a set system, where each set (or block) has the same cardinality. It turns out that examining designs that have specific structure is especially useful in the distributed computing context. In this work, we will make use of *resolvable designs*, a special category of block designs.

**Definition 3.** A subset $\mathcal{P} \subset X$ in a design $(X, \mathcal{A})$ is said to be a *parallel class* if for $X_i \in \mathcal{P}$ and $X_j \in \mathcal{P}$ with $i \neq j$ we have $X_i \cap X_j = \emptyset$ and $\cup_{\{j:X_j \in P\}} X_j = X$. A partition of $\mathcal{A}$ into several parallel classes is called a resolution, and $(X, \mathcal{A})$ is said to be a resolvable design if $\mathcal{A}$ has at least one resolution.

A simple example of a resolvable design is obtained by considering all 2-subsets of $\{1, \ldots, 4\}$.

**Example 3.** Let $X = \{1,2,3,4\}$ and $\mathcal{A} = \{\{1,2\},\{3,4\},\{1,3\},\{2,4\},\{1,4\},\{2,3\}\}$. The $(X,\mathcal{A})$ forms a resolvable design with the following parallel classes.

$$\mathcal{P}_1 = \{\{1,2\},\{3,4\}\},$$
$$\mathcal{P}_2 = \{\{1,3\},\{2,4\}\}, \text{ and}$$
$$\mathcal{P}_3 = \{\{1,4\},\{2,3\}\}.$$

We note here that Example 2 used precisely this design, when specifying the subpacketization and the placement.

It turns out that there is a systematic procedure for constructing resolvable designs, where the starting point is an error correcting code [17]. We explain this procedure below.

Let $\mathbb{Z}_q$ denote the additive group of integers modulo $q$. The generator matrix of an $(k, k-1)$ single parity-check (SPC) code over $\mathbb{Z}_q$[1] is defined by

$$\mathbf{G}_{SPC} = \left[\begin{array}{c|c} & 1 \\ \mathbf{I}_{k-1} & \vdots \\ & 1 \end{array}\right]. \tag{1}$$

This code has $q^{k-1}$ codewords. The codewords are $\mathbf{c} = \mathbf{u} \cdot \mathbf{G}_{SPC}$ for each possible message vector $\mathbf{u}$. The code is systematic so that the first $k-1$ symbols of each codeword are the same as the symbols of the message vector. The $q^{k-1}$ codewords $\mathbf{c}_i$ computed in this manner are stacked into the columns of a matrix $\mathbf{T}$ of size $k \times q^{k-1}$.

$$\mathbf{T} = [\mathbf{c}_1^T, \mathbf{c}_2^T, \cdots, \mathbf{c}_{q^{k-1}}^T]. \tag{2}$$

The corresponding resolvable design is constructed as follows. Let $X_{SPC} = [q^{k-1}]$ (we use $[n]$ to denote the set $\{1, 2, \ldots, n\}$ throughout) represent the point set of the design. We define the blocks as follows. For $0 \leq l \leq q-1$, let $B_{i,l}$ be a block defined as

$$B_{i,l} = \{j : \mathbf{T}_{i,j} = l\}.$$

The set of blocks $\mathcal{A}_{SPC}$ is given by the collection of all $B_{i,l}$ for $1 \leq i \leq k$ and $0 \leq l \leq q-1$ so that $|\mathcal{A}_{SPC}| = kq$. The following lemma (see [15, Appendix] for proof) shows that this construction always yields a resolvable design.

**Lemma 1.** The above scheme always yields a resolvable design $(X_{SPC}, \mathcal{A}_{SPC})$ with $X_{SPC} = [q^{k-1}]$, $|B_{i,l}| = q^{k-2}$ for all $1 \leq i \leq k$ and $0 \leq l \leq q-1$. The parallel classes are analytically described by $\mathcal{P}_i = \{B_{i,l} : 0 \leq l \leq q-1\}$, for $1 \leq i \leq k$.

**Example 4.** The generator matrix of this $(3,2)$ SPC code over $\mathbb{Z}_2$ (binary) is given by

$$\mathbf{G}_{SPC} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

The matrix $\mathbf{T}$ can be obtained as

$$\mathbf{T} = [\mathbf{c}_1^T, \mathbf{c}_2^T, \mathbf{c}_3^T, \mathbf{c}_4^T] = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

[1]We emphasize that this construction works even if $q$ is not a prime, i.e., $\mathbb{Z}_q$ is not a field.

**Algorithm 1** Proposed Protocol

1: Input: File $\mathcal{W}$, $Q$ functions; number of servers $K = k \times q$. $K$ divides $Q$.
2: Use a $(k, k-1)$ SPC code to generate a design $(\mathcal{X}, \mathcal{A})$
3: Split $\mathcal{W}$ into $q^{k-1}$ disjoint subfiles, $w_1, \ldots, w_{q^{k-1}}$.
4: Assign subfiles to servers such that server $B_{i,j}$ is assigned subfile $w_\ell$ if $\ell \in B_{i,j}$.
5: Choose an equal-size partition of $[Q]$ to obtain the sets $\phi^{B_{i,j}}$ for $i = 1, \ldots, k$ and $j = 0, \ldots, q-1$.
6: Execute the Map phase on each of the servers.
7: Choose all possible sets $\{B_{1,j_1}, B_{2,j_2}, \ldots, B_{k,j_k}\}$ where $j_\ell \in \{0, \ldots, q-1\}$, such that $\cap_{\ell=1}^{k} B_{\ell,j_\ell} = \emptyset$ and store them in a collection $\mathcal{G}$.
8: **for** $\gamma \in [Q/K]$ **do**
9:  **for** each group $G = \{B_{1,j_1}, B_{2,j_2}, \ldots, B_{k,j_k}\} \in \mathcal{G}$ **do**
10:  Determine $\Delta_\ell^G = \nu_{\phi^{B_{i,j}}[\gamma], \cap_{k \neq \ell} B_{k,j_k}}$ for $\ell = 1, \ldots, k$.
11:  Split packet $p(\Delta_\ell^G)$ into $k-1$ parts at each server where it is available.
12:  Label each split of $p(\Delta_\ell^G)$, arbitrarily with distinct labels from $1, \ldots, k-1$.
13:  For each $\ell \in [k]$, server $B_{\ell,j_\ell}$ transmits
$$\bigoplus_{m \neq \ell} p(\Delta_m^G)[\text{label}(\Delta_m^G, B_{\ell,j_\ell})]$$
14:  **end for**
15: **end for**
16: Execute Reduce phase on each of the servers.

It can be observed, e.g., that $B_{1,0} = \{1, 2\}$ and $B_{1,1} = \{3, 4\}$ so that they form a parallel class. In fact, this construction returns the resolvable design considered in Example 3.

### A. From resolvable designs to protocol specification

The main idea in our work is to use an appropriate resolvable design to specify the number of subfiles, the map task assignments and the messages transmitted in the shuffle phase for a given distributed computing job. We explain this correspondence next.

Consider a file $\mathcal{W}$ on which $Q$ functions need to be computed and suppose that we have access to $K$ servers; we assume that $Q$ is a multiple of $K$. In Algorithm 1, we present the steps that specify the entire protocol. The protocol can be understood as follows. We choose an integer $q$ such that $q$ divides $K$, i.e., $K = k \times q$. Next, we form a $(k, k-1)$ SPC code and the corresponding resolvable design using the aforementioned procedure. The point set $\mathcal{X} = [q^{k-1}]$ and the block set $\mathcal{A}$ will be such that $|\mathcal{A}| = kq$. The blocks of $\mathcal{A}$ will be indexed as $B_{i,j}, i = 1, \ldots, k$ and $j = 0, 1, \ldots, q-1$.

We associate the point set $\mathcal{X}$ with the subfiles, i.e., $N = |\mathcal{X}| = q^{k-1}$ and the block set $\mathcal{A}$ with the servers. The map task assignment follows the natural incidence between the points and the blocks, i.e., server $B_{i,j}$ is responsible for executing the map tasks on the set of subfiles $\text{Map}[B_{i,j}] = \{w_\ell \mid \ell \in B_{i,j}\}$. Thus, at the end of the map phase, server $B_{i,j}$ has computed the $Q$ intermediate value on the subfiles in $\text{Map}[B_{i,j}]$.

Recall that we assume that $K$ divides $Q$. Thus, each server is responsible for reducing $Q/K$ functions. We let $\phi^{B_{i,j}} \subset [Q]$ represent the set of functions assigned for reduction to server $B_{i,j}$. The sets $\phi^{B_{i,j}}$ form a partition of $[Q]$. For ease of notation, we let $\phi^{B_{i,j}}[\ell]$ represent the $\ell$-th function in the set $\phi^{B_{i,j}}$; $\ell$ ranges from 1 to $Q/K$.

Following the map phase, in the shuffle phase, each server $B_{i,j}$ needs intermediate values from other servers so that it has enough information to reduce the functions in $\phi^{B_{i,j}}$. In this step we transmit coded packets that are simultaneously useful to multiple servers. Towards this end we form a collection of user groups by choosing one block from each parallel class according to the rule in Step 7 of the protocol, i.e., we choose servers $B_{1,j_1}, B_{2,j_2}, \ldots, B_{k,j_k}$ such that $\cap_{\ell=1}^{k} B_{\ell,j_\ell} = \emptyset$. For a given user group $G$ (of size $k$) we can show that each server in $G$ can transmit a useful message, denoted $\Delta_\ell^G$, to $k-1$ other servers where $\ell \in \{1, \ldots, k\}$ represents the label of the transmitting server. Each server in $G$ is assigned a unique label according to a bipartite graph (see [15, Section III.C] for details). Furthermore, we can show that considering all possible user groups allows the shuffle phase to achieve its objective, i.e., at the end of the shuffle phase, all servers have enough information to execute the reduce phase. An instance of the shuffle phase equations was discussed in Example 2 (see Table I).

### B. Communication load analysis

We now present an analysis of the achieved communication rate. The proof of correctness of our protocol and a detailed communication load analysis can be found in [15, Section III.C]. In the uncoded case, each server needs $QN/K$ intermediate values $\nu_{j,n}$'s to execute its reduce phase. Note that each server already has $rN/K \times Q/K$ of them. Thus, in the shuffle phase the communication load is given by

$$\begin{aligned} L_{uncoded} &= \frac{K(QN/K - rQN/K^2)B}{QNB} \\ &= 1 - \frac{r}{K}. \end{aligned}$$

On the other hand, for our scheme, the number of bits transmitted in shuffle phase is given by

$$q^{k-1}(q-1) \cdot B \frac{k}{k-1} \cdot \frac{Q}{K}.$$

Thus, the communication load is given by

$$\begin{aligned} L_{prop} &= \frac{q^{k-1}(q-1) \cdot B \frac{k}{k-1} \cdot \frac{Q}{K}}{QNB} \\ &= \frac{1}{k-1}\left(1 - \frac{k}{K}\right), \end{aligned}$$

where the second equality above is obtained by using the fact that $N = q^{k-1}$ and $K = kq$.

Next, note that for our proposed scheme the computation load is $k$, i.e., $r = k$. Thus, we reduce the overall communication load by a factor of $\frac{1}{r-1}$ with respect to an uncoded system. In contrast, the approach of [7], reduces the communication load by a factor of $\frac{1}{r}$. However, this comes at the expense of a large $N$ as discussed previously.

TABLE II
SPECIFICATIONS OF THE AMAZON EC2 MACHINES OF THE CLUSTER

| Machine role | Instance type | Virtual CPUs | Memory | Storage |
|---|---|---|---|---|
| Master | r3.large | 2 | 15.25GB | 32GB SSD |
| Server | m3.large | 2 | 7.5GB | 32GB SSD |

## IV. DETAILS OF IMPLEMENTATION

We have implemented TeraSort on Amazon EC2 clusters using our proposed approach. The implementation was performed in C++ using the Open MPI library for communication among the processes of the master and the servers. Our code builds on [18] and comparisons with the uncoded case and the approach of [8] have been made.

### A. Data set description

We used a 12GB data set such that each line of the file is a key-value (KV) pair consisting of an integer key and a arbitrary string value. The sorting is done based on the key. It is not too hard to see that this KV formulation can be put in on-to-one correspondence with the formulation in terms of map and reduce functions (*cf.* Section II).

### B. Amazon EC2 cluster configuration

We used Amazon EC2 instances among which one served as a master and the rest of them as slaves (servers). The specifications of these machines are given in Table II. After placing the subfiles to the carefully chosen servers we also impose a limit of 100Mbps for both incoming and outgoing traffic of all machines[2]. This serves the purpose of alleviating bursty TCP transmissions.

### C. Platform and code implementation description

The source code is in C++ and we used the Open MPI library, version 1.10.2, for communication among the processes of the master and the servers. Our source code repository is available at [19].

The overall sequence of steps in processing a given job are: CodeGen $\rightarrow$ Map $\rightarrow$ Pack/Encode $\rightarrow$ Shuffle $\rightarrow$ Unpack/Decode $\rightarrow$ Reduce. We explain these steps below.

- *Code generation*: All nodes (including the master) start by generating the resolvable design based on our choice of the parameters $q$ and $k$. Next, the data set is split into $N$ subfiles by the master and the appropriate subfiles are transmitted to each slave based on Step 4 of the protocol.
- *Map.* For each subfile $w_a$ that server $B_{i,l}$ has in its block, it will compute $\{\nu_{1,a}, \ldots, \nu_{Q,a}\}$ during the Map phase.
- *Pack/Encode.* For the uncoded implementation, we use the Pack operation. The Pack stage stores all intermediate values that will be sent to the same reducer in a continuous memory array so that a single TCP connection for each sender/receiver pair suffices (which may transmit multiple KV pairs) when `MPI_Send` is called[3].

  In the coded implementation encoded packets are created from the mapped data as described in Algorithm 1.
- *Shuffle.* For each shuffling group $G$ a server belongs to, it will broadcast an appropriate encoded packet to the rest of the group.
- *Unpack/Decode.* In the uncoded implementation we use the Unpack operation which simply deserializes the received data to a list of KV pairs. In the coded implementation the intermediate values are decoded locally on each server from the received data.
- *Reduce.* The reduce function is applied on the unpacked/decoded data.

## V. RESULTS AND DISCUSSION

### A. Experimental Results

Tables III and IV contain the results of TeraSort using our approach and comparisons with the approach of [8]. There are approximately $130 \times 10^6$ KV pairs to be sorted. Table III presents the time each phase needed to completed including the memory allocation time. We have included a column for the total time that omits the memory allocation cost and another column that shows the speedup in that case. Table IV presents only the memory allocation time of certain MapReduce phases for the same experiments. The need to take into account the memory allocation cost comes from the fact that for data sets at this scale, dynamic memory allocation on the heap has a non-negligible impact on the total time. We note here that the results in [8] are generated using the code in [18] which explicitly ignores the memory allocation time (*cf.* communication with the first author of [8]). We emphasize however, that the results in Table III indicate that our approach is consistently superior whether or not one takes into account the memory allocation time.

To understand the effect of choosing different values of $N$, we applied our algorithm with different values of $(k, q) = (r, q)$ pairs.

We observe from Table III that if we account for memory allocation cost, our scheme achieves up to $4.69\times$ speedup compared to the uncoded TeraSort whereas if we ignore this code our schemes demonstrates an improvement of up to $5.51\times$. Moreover, the gain over the prior coded TeraSort scheme, if we compare the best time reported by each scheme, can go up to $4.69/1.56 \approx 3.01\times$ (when including memory allocation time) or $5.51/2.30 \approx 2.4\times$ (when excluding memory allocation time). According to the assumptions of our scheme, the possible values of $r$ for $K = 16$ servers are $1, 2, 4$ and $8$ but based on the rate analysis of Section III-B only $4$ and $8$ would yield a gain which is the reason we chose those values. We note here that the shuffle phase results corresponding to $r = 8$ for prior work could not be obtained as their program crashed. A more detailed discussion about Tables III and IV can be found in [15, Section V.A].

### B. Discussion

The major issue of the prior scheme [8] is the large value of $N$ that it needs. This translates into a large number ($\binom{K}{r+1}$) of user groups in the shuffling phase.

[2]In order to manipulate the traffic control settings, we use the Linux `tc` command

[3]In the Shuffling phase of the uncoded case, each server unicasts data to a single receiver at any particular time, which is exactly the purpose of `MPI_Send` call.

TABLE III
MAPREDUCE TIME FOR SORTING 12GB DATA ON 16 SERVER NODES INCLUDING THE MEMORY ALLOCATION COST

| | CodeGen (sec.) | Map (sec.) | Pack/ Encode (sec.) | Shuffle (sec.) | Unpack/ Decode (sec.) | Reduce (sec.) | Total Time (sec.) | | Speedup | | Rate (Mbps.) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | w/MA | w/out MA | w/MA | w/out MA | |
| Uncoded: | - | 5.71 | 11.75 | 1105.64 | 4.46 | 12.88 | 1140.44 | 1126.68 | | | 100.83 |
| Prior: $r = 3$ | 5.82 | 17.94 | 229.80 | 455.05 | 6.23 | 14.54 | 729.38 | 496.76 | $1.56\times$ | $2.27\times$ | 64.79 |
| Prior: $r = 5$ | 26.78 | 29.99 | 1000.15 | 297.28 | 8.16 | 16.47 | 1378.83 | 490.28 | $0.83\times$ | $2.30\times$ | 61.04 |
| Prior: $r = 8$ | 38.41 | 51.03 | 1128.16 | - | - | - | - | - | - | - | - |
| Proposed: $r = 4$ | 0.64 | 25.91 | 9.93 | 307.15 | 6.91 | 17.29 | 367.83 | 352.91 | $3.10\times$ | $3.19\times$ | 88.47 |
| Proposed: $r = 8$ | 0.61 | 62.46 | 26.22 | 127.43 | 8.38 | 17.85 | 242.95 | 204.58 | $4.69\times$ | $5.51\times$ | 62.68 |

TABLE IV
MEMORY ALLOCATION COST FOR SORTING 12GB DATA ON 16 SERVER NODES

| | Map (sec.) | Pack/Encode (sec.) | Unpack/Decode (sec.) | Total Time (sec.) |
|---|---|---|---|---|
| Uncoded: | 2.32 | 9.17 | 2.27 | 13.76 |
| Prior: $r = 3$ | 6.87 | 223.16 | 2.59 | 232.62 |
| Prior: $r = 5$ | 11.29 | 874.14 | 3.12 | 888.55 |
| Prior: $r = 8$ | 18.03 | 968.48 | - | - |
| Proposed: $r = 4$ | 9.91 | 1.85 | 3.16 | 14.92 |
| Proposed: $r = 8$ | 22.01 | 13.17 | 3.19 | 38.37 |

This number can be prohibitive for today's High Performance Computing (HPC) communication protocols like the Message Passing Interface (MPI). This is because all MPI communication is associated with a *communicator* that describes the communication context and an associated group of processes, as seen in [20]. But, the cost of splitting the initial communicator into smaller communicators each of which facilitates the communication within a group is non-negligible [21]. Specifically, based on the experiments we performed on our cluster for this scheme of splitting the cost is exponential in $r$ and can easily dominate the overall MapReduce execution.

Another point to consider is that most MPI libraries support a limited number of communicators that can easily be exceeded by $\binom{K}{r+1}$, even for relatively small numbers of $K$ and $r$, thus rendering the prior scheme impossible to implement.

More specifics on the above issues and some of our experiments that illustrate them can be found in [15, Section V.B].

## VI. DIRECTIONS FOR FUTURE WORK

For the purposes of this paper the cluster was assumed homogeneous, i.e., consisting of similar servers connected via an error-free shared link. An interesting extension would be to examine potential benefits of resolvable designs on heterogeneous clusters where the nodes have potentially different storage, memory and processing capabilities.

## REFERENCES


[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] "Apache Hadoop." [Online]. Available: http://hadoop.apache.org/

[3] "Apache Spark." [Online]. Available: http://spark.apache.org

[4] A. Cuzzocrea, I.-Y. Song, and K. C. Davis, "Analytics over Large-scale Multidimensional Data: The Big Data Revolution!" in *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP*, ser. DOLAP '11, 2011, pp. 101–104.

[5] 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)*, San Jose, CA, 2013, pp. 107–117.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 98–109, Aug. 2011.

[7] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A Fundamental Tradeoff Between Computation and Communication in Distributed Computing," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, Jan 2018.

[8] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and S. Avestimehr, "Coded TeraSort," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 389–398.

[9] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," *IEEE Transactions on Information Theory*, vol. 60, no. 5, pp. 2856–2867, May 2014.

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

[11] L. Tang and A. Ramamoorthy, "Coded caching schemes with reduced subpacketization from linear block codes," *IEEE Trans. on Info. Th.*, vol. 64, no. 4, pp. 3099–3120, Apr. 2018.

[12] O. Olmez and A. Ramamoorthy, "Fractional repetition codes with flexible repair from combinatorial designs," *IEEE Trans. on Info. Th.*, vol. 62, no. 4, pp. 1565 –1591, 2016.

[13] A. S. Tripathy and A. Ramamoorthy, "Sum-networks from incidence structures: construction and capacity analysis," *IEEE Trans. on Info. Th.*, vol. 64, no. 4, pp. 3461–3480, May 2018.

[14] L. Tang and A. Ramamoorthy, "Coded caching for networks with the resolvability property," in *IEEE Intl. Symposium on Info. Th.*, 2016.

[15] K. Konstantinidis and A. Ramamoorthy, "Leveraging Coding Techniques for Speeding up Distributed Computing," 2017 [Online] Available: https://arxiv.org/pdf/1802.03049.

[16] D. R. Stinson, *Combinatorial Designs: Constructions and Analysis*. Springer, 2004.

[17] S. Lin and D. J. Costello, *Error Control Coding, 2nd Ed.* Prentice Hall, 2004.

[18] "Repository of TeraSort for prior implementation." [Online]. Available: https://github.com/AvestimehrResearchGroup/Coded-TeraSort/tree/IgnoreMemoryTime

[19] "SPC Coded TeraSort repository." [Online]. Available: https://bitbucket.org/kkonstantinidis/codedterasort

[20] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI (3rd Ed.): Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 2014.

[21] P. Sack and W. Gropp, *A Scalable MPI_Comm_split Algorithm for Exascale Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–10.