Coded Distributed Computing: Fundamental Limits and Practical Challenges

Songze Li*, Qian Yu*, Mohammad Ali Maddah-Ali †, and A. Salman Avestimehr*

* Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA
† Nokia Bell Labs, Holmdel, NJ, USA

Abstract—In this paper, we demonstrate a coded computing framework, named Coded Distributed Computing (CDC), which optimally trades extra computation resources for communication bandwidth in a MapReduce-type distributed computing environment. We also empirically illustrate the practical impact of CDC by analyzing the performance of a distributed sorting algorithm, named CodedTeraSort, which was developed by integrating the coding principle of CDC into the Hadoop benchmark TeraSort. Experiment results illustrate $1.97 \times - 3.39 \times$ speedup using CodedTeraSort, compared with TeraSort, for typical settings of interest. In the end, we review some of the open problems and future directions.

I. INTRODUCTION

We consider a general distributed computing framework, motivated by a widely used computing primitive MapReduce [1], in which the overall job execution is decomposed into two stages: “Map” and “Reduce”. Firstly in the Map stage, distributed computing nodes process parts of the input data locally, generating some intermediate values according to their designed Map functions. Next, they exchange the calculated intermediate values among each other (a.k.a. data shuffling), in order to calculate the final output results distributedly using their designed Reduce functions.

The data shuffling process often appears to limit the performance of MapReduce applications. For example, in a Facebook’s Hadoop cluster, it is observed that 33% of the overall job execution time is spent on data shuffling [2]. Also as is observed in [3], 70% of the overall job execution time is spent on data shuffling when running Self-Join on an Amazon EC2 cluster. Recently in [4]–[6], coding was introduced into distributed computing, in order to reduce the load of shuffling intermediate results across computing nodes, hence speeding up the overall computation. A general MapReduce-type distributed computing model was introduced in [6], in which it was demonstrated that coding can be applied on both Map task placement and data shuffling, significantly slashing the load of communication.

Within this model, a tradeoff between the communication load (normalized total number of shuffled bits) and the computation load (normalized total number of computed Map functions) was formalized and exactly characterized in [6]. In particular, for a distributed computing application run on $K$ servers and a computation load of $r$, $r \in \{1, \ldots, K\}$, the minimum required communication load was characterized as $L^*(r) = \frac{1}{r} \cdot (1 - \frac{1}{r}) = \Theta(\frac{1}{r})$, revealing a fundamental inverse-linearly proportional relationship between the computation and the communication loads. In particular, a coded computing framework, named Coded Distributed Computing (CDC), was proposed in [6] to exactly achieve this tradeoff. CDC utilizes a carefully designed repetitive placement of the Map tasks at $r$ distinct servers, creating coded multicast messages that simultaneously satisfy the data demands of $r$ servers. Hence, compared with an uncoded data shuffling scheme, CDC reduces the communication load by exactly a factor of the computation load $r$.

In this paper, we first provide an overview of the CDC scheme by 1) illustrating its key coding techniques via a simple example, and 2) discussing the results that CDC optimally trades computation resources for communication bandwidth. Then we study the practical impact of the CDC scheme on the total execution times of distributed computing applications. To do that, we analyze the empirical evaluations of the performance of a novel distributed sorting algorithm, named CodedTeraSort, which was developed by integrating the coding principle of CDC into a widely used Hadoop benchmark TeraSort [7]. From extensive experiments on Amazon EC2 clusters, we observe that by performing $3 \times$ or $5 \times$ more computations in the Map stage, CodedTeraSort achieves $1.97 \times - 3.39 \times$ speedup, compared with TeraSort.

Finally, we conclude the paper by discussing some open problems on how to utilize the coding techniques of CDC to achieve efficient distributed computing in more general settings. Specifically, these problems involve dealing with heterogeneous computing nodes, multi-stage computation tasks, networks with multi-layer and structured topology, coding for Edge/Fog computing, and exploiting the algebraic structures of computation tasks.

II. OVERVIEW OF CODED DISTRIBUTED COMPUTING (CDC)

In this section, we illustrate the recently proposed Coded Distributed Computing (CDC) framework [4]–[6] via an example, to illustrate how coding can be utilized to reduce the data shuffling load in distributed computing. We also review the results in [6] on characterizing the exact tradeoff between the computation load and the communication 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 distributedly 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.

The driving idea for CDC 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. We next illustrate CDC via a simple example.

Example (Coded Distributed Computing). 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, 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 $4 \times 3 = 12$.

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. 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.

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 $q = 1, \ldots, Q$:

$$\phi_q(x_1, \ldots, x_N) = g_q(\ell_{q,1}(x_1), \ldots, \ell_{q,N}(x_N)).$$

(1)

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

In the general CDC scheme proposed in [4, 6], 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 other nodes. As a result, CDC reduces the communication load by exactly a multiplicative factor of the computation load $r$ (see Fig. 2). Also in [6], an information-theoretic lower bound on the minimum possible communication load, $L^*(r)$, was derived and stated in the following theorem, which exactly matches that achieved by CDC.

**Theorem 1.** The minimum possible communication load, for a computation load $r \in \{1, \ldots, K\}$, is characterized by

$$L^*(r) = L_{\text{coded}}(r) \triangleq \frac{1}{r} \cdot \left(1 - \frac{1}{r}ight) = \Theta\left(\frac{1}{r}\right),$$

(2)

for sufficiently large $N$. For general $1 \leq r \leq K$, $L^*(r)$ is the lower convex envelop of the above points.

\footnote{Note that when a node maps a file, it computes all three intermediate values of that file needed for the three output functions.}

\footnote{This type of coding was also utilized to solve the index coding problem [8, 9] that arises from the network coding problem [10].}
Theorem 1 has revealed a fundamental inversely-linear proportional tradeoff 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.

In [6], the CDC scheme was also generalized to tackle a “cascaded distributed computing framework”, in which each output function is computed by $s$ nodes, for some $s \in \{1, \ldots, K\}$. The minimum possible communication load for the cascaded framework, $L^*(r, s)$, which is achieved by a generalized CDC scheme, was stated in the following theorem.

**Theorem 2.** The minimum possible communication load for the cascaded framework, in which each output function is computed by $s \in \{1, \ldots, K\}$ nodes, is characterized by

$$L^*(r, s) = L_{\text{coded}}(r, s) \triangleq \sum_{\ell = \max(r+1, s)}^{r+s} \frac{\ell(K) \left(\frac{r}{\ell} \right) + (r-s)}{r(K) \left(\frac{r}{r-s} \right)},$$

for sufficiently large $Q$ and $N$, and $r \in \{1, \ldots, K\}$. For general $1 \leq r \leq K$, $L^*(r, s)$ is the lower convex envelop of the above points $\{(r, L_{\text{coded}}(r, s)) : r \in \{1, \ldots, K\}\}$.

**Remark 1.** The idea of structured Map task placement to enable efficient coded multicasting in CDC was initially proposed in the context of cache networks in [11], [12], and extended in [13] for wireless D2D networks, where caches pre-fetch parts of the content in a structured way to enable coding during the content delivery, minimizing the network traffic.

### III. Practical Impact

Having theoretically demonstrated the optimality of the Coded Distributed Computing (CDC) scheme in trading computing resources for network bandwidth, we are interested in the practical impact of CDC on improving the performance of distributed computing workloads, e.g., total execution time. In this section, we first analyze the total execution time of a MapReduce job using CDC. We then focus on the “sorting” application, and empirically demonstrate the performance of a coded distributed sorting algorithm, named CodedTeraSort, which was developed by applying the principle of CDC into the Hadoop MapReduce benchmark TeraSort [7].

#### A. Job Execution Time of CDC

Within the MapReduce framework, utilizing the CDC scheme can reduce the overall job 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 $T_{\text{map}}$, the time spent shuffling intermediate values, denoted by $T_{\text{shuffle}}$, and the time spent executing the Reduce tasks, denoted by $T_{\text{reduce}}$, i.e.,

$$T_{\text{total, MR}} = T_{\text{map}} + T_{\text{shuffle}} + T_{\text{reduce}}.$$  

Using CDC, 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, CDC promises that we can achieve the overall execution time of

$$T_{\text{total, CDC}} \approx rT_{\text{map}} + \frac{1}{r}T_{\text{shuffle}} + T_{\text{reduce}},$$

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

$$r^* = \sqrt[4]{\frac{T_{\text{shuffle}}}{T_{\text{map}}}} \quad \text{or} \quad \sqrt[4]{\frac{T_{\text{map}}}{T_{\text{shuffle}}}},$$

resulting in execution time of

$$T_{\text{total, CDC}} \approx 2\sqrt[4]{\frac{T_{\text{shuffle}}}{T_{\text{map}}}} T_{\text{map}} + T_{\text{reduce}}.$$  

For example, in a MapReduce application that $T_{\text{shuffle}}$ is $10 \times - 100 \times$ larger than $T_{\text{map}} + T_{\text{reduce}}$, by comparing from (4) and (6), we note that CDC can reduce the execution time by approximately $1.5 \times - 5 \times$.

Next, we empirically demonstrate the impact of coding on speeding up a distributed sorting algorithm CodedTeraSort, which was developed by integrating the coding techniques of CDC into a conventional MapReduce sorting algorithm TeraSort.

#### B. Coded Distributed Sorting

1) **Algorithm:** TeraSort [14] is a distributed algorithm for sorting a large amount of data. The input data that is to be sorted is in the format of key-value (KV) pairs. 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.

In a conventional TeraSort implementation over $K$ distributed computing nodes, the entire input KV pairs are partitioned into $K$ files, each of which is placed on a unique node. The domain of the keys is also split into $K$ ordered partitions (e.g., consecutive intervals of integers), and each node will be responsible for sorting the KV pairs whose keys fall into one of the partitions. In the Map stage, each node hashes each KV pair in its local file into one of the $K$ partitions, according to its key. In the Shuffle stage, the KV pairs in the same partition are delivered to the node that is responsible for sorting that
partition. In the Reduce stage, each node locally sorts KV pairs belonging to its assigned partition. A simple example illustrating TeraSort is shown in Fig. 3.

**Fig. 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.

In [15], a novel distributed sorting algorithm, named CodedTeraSort, was proposed by applying the coding techniques of CDC into TeraSort. The key idea of CodedTeraSort is that for some design parameter \( r \in \mathbb{N} \), each input KV pair is placed and processed on \( r \) computing nodes, according to a structure specified by the file placement strategy of CDC. Then coded multicast packets can be created in the Shuffle stage to simultaneously deliver required KV pairs to \( r \) nodes, saving the bandwidth consumption by \( r \times \). Hence, the overall execution time of CodedTeraSort can be roughly expressed by (5).

**2) Performance Evaluation:** Extensive experiments on Amazon EC2 clusters were performed in [15], in order to evaluate the performance of TeraSort and CodedTeraSort. In particular, a system architecture that consists of a coordinator node and \( K \) worker nodes, for some \( K \in \mathbb{N} \), was employed. 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. 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 workers’ memories, all intermediate data that are used for encoding, decoding and local sorting are persisted in the workers’ memories, and hence there is no disk I/O involved during the executions of the algorithms.

Communications among EC2 instances were implemented using Open MPI library [16]. In TeraSort, packets were unicast between worker instances using MPI_Send. In CodedTeraSort, coded packets were multicast to multiple workers using MPI_Bcast. For both algorithms, the data shuffling were performed serially, i.e., only one worker is communicating at any time instance. In this particular implementation, a Pack stage and an Unpack stage were added to separate out the time spent to serialize and deserialize the communicated KV pairs respectively.

Compared with TeraSort, three stages CodeGen, Encode, and Decode were added to the implementation of CodedTeraSort. In the CodeGen (or code generation) stage, each node uses MPI_Comm_split to initialize \( (K+1) \) multicast groups each containing \( r+1 \) nodes on Open MPI, such that multicast communications will be performed within each of these groups. In the Encode stage, each node creates coded multicast packets that are simultaneously useful for \( r \) other nodes. In the Decode stage, each node decodes the required KV pairs using the local computing results from the Map stage.

A part of the experiment results are shown here in Tables I and II. We observe an overall \( 1.97 \times 3.39 \) speedup of CodedTeraSort as compared with TeraSort. From the experiment results we make the following observations:

- The total execution time of CodedTeraSort improves over TeraSort whose Shuffle time dominates the computation times of the other stages.
- For CodedTeraSort, the time spent in the CodeGen stage is proportional to \( (K+1) \), 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.
- 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 facts. 1) The MPI_Bcast API has an inherent overhead for multicast, for instance, a multicast tree is constructed before multicasting to a set of nodes. 2) Using MPI_Bcast, the time of multicasting a packet to \( r \) nodes is higher than that of unicasting the same packet to a single node.

Finally, 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 execution time increases linearly with \( r \), and more importantly the CodeGen time increases as \( (K+1) \). Hence, for small values of \( r (r < 6) \), we observe overall reduction in execution time, and the speedup increases. However, as \( r \) increases, the CodeGen time will dominate the execution time, and the speedup decreases. Hence, in the evaluations, \( r \) was limited to be at most 5.

The impact of worker number \( K \): As \( K \) increases, the speedup decreases. This is due to the following two reasons. 1) The computation complexity of the CodeGen stage grows exponentially with \( K \). 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.

\[ The \ redundancy \ parameter \ r \ is \ also \ limited \ by \ the \ total \ storage \ available \ at \ the \ nodes. \ Since \ for \ a \ choice \ of \ redundancy \ parameter \ r, \ each \ piece \ of \ input \ KV \ pairs \ should \ be \ stored \ at \ r \ nodes, \ and \ r \ cannot \ be \ increased \ beyond \ total \ available \ storage \ at \ the \ worker \ nodes. \]
IV. CONCLUDING REMARKS AND OPEN PROBLEMS

We illustrated the Coded Distributed Computing (CDC) framework that optimally trades extra computation resources in the network for communication bandwidth. We empirically demonstrated the impact of CDC by analyzing the execution time of a basic distributed sorting algorithm CodedTeraSort, which was developed by integrating the coding principle of CDC into a conventional algorithm TeraSort. We conclude the paper by highlighting three interesting future research directions and open problems.

Heterogeneous Computing Nodes. It is common to have computing nodes with different storage, processing and communication capacities within computer clusters. In fact, straggling servers can severely degrade the performance of distributed computing applications. Recently in [17], Maximum-Distance-Separable (MDS) codes were utilized to encode linear computation tasks, providing robustness to a certain number of stragglers. A unified coding scheme that superimposes the CDC scheme on top of MDS codes was proposed in [13], which achieves a flexible tradeoff between computation latency and communication load when faced with straggling servers. Nevertheless, developing practical resource allocation strategies and coding techniques for general heterogeneous distributed computing environments, that are provably optimum (approximately), is a challenging open problem.

Multi-Stage Computation Tasks. Another important direction is to consider more general computing frameworks, in which the computation job is represented in the form of a Directed Acyclic Task Graph (DAG). While we can apply the generalized CDC scheme (see Theorem 2) for each stage of computation locally, we expect to achieve a higher reduction in bandwidth consumption and response time by globally trading extra computation resources scattered across the network for communication bandwidth. A preliminary exploration along this direction was recently presented in [19].

Edge/Fog Computing. In the Edge/Fog computing paradigm, abundant computation resources scattered across the network edge (e.g., smartphones, tablets and smart cars) are harvested to perform data-intensive computations collaboratively. In this scenario, coding opportunities are widely available by injecting redundant storage and computations into the edge network. We envision codes to play a transformational role in Edge/Fog computing for leveraging such redundancy to substantially reduce the bandwidth consumption and the latency of computing. In [20], we have incorporated the coding techniques of CDC into an edge-facilitated wireless distributed computing platform, achieving a scalable design such that the platform can accommodate an unlimited number of mobile users with a constant amount of bandwidth consumption.

REFERENCES