2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)
Allerton Park and Retreat Center
Monticello, IL, USA, September 24-27, 2019

Straggler Resilient Serverless Computing Based on Polar Codes

Burak Bartan and Mert Pilanci
Department of Electrical Engineering, Stanford University
Email: {bbartan, pilanci}@stanford.edu

***Abstract*—We propose a serverless computing mechanism for distributed computation based on polar codes. Serverless computing is an emerging cloud based computation model that lets users run their functions on the cloud without provisioning or managing servers. Our proposed approach is a hybrid computing framework that carries out computationally expensive tasks such as linear algebraic operations involving large-scale data using serverless computing and does the rest of the processing locally. We address the limitations and reliability issues of serverless platforms such as straggling workers using coding theory, drawing ideas from recent literature on coded computation. The proposed mechanism uses polar codes to ensure straggler-resilience in a computationally effective manner. We provide extensive evidence showing polar codes outperform other coding methods. We have designed a sequential decoder specifically for polar codes in erasure channels with full-precision input and outputs. In addition, we have extended the proposed method to the matrix multiplication case where both matrices being multiplied are coded. The proposed coded computation scheme is implemented for AWS Lambda. Experiment results are presented where the performance of the proposed coded computation technique is tested in optimization via gradient descent. Finally, we introduce the idea of partial polarization which reduces the computational burden of encoding and decoding at the expense of straggler-resilience.**

I. Introduction

Computations for large-scale problems on the cloud might require a certain level of expertise for managing servers, and certainly requires configuration of resources ahead of time. It might be challenging to estimate beforehand exactly how much RAM, storage, and time a certain problem requires. For server-based computing on the cloud, users have to manage these configurations which if not done right, costs more money for users. Serverless computing, on the other hand, emerges as a platform simpler to use with much less overhead. Serverless functions offer an alternative to large-scale computing with a lot to offer in terms of cost and speed.

In this work, we present a mechanism based on serverless computing that enables users to run their computations in their computers except whenever there is a large-scale matrix multiplication, the computation is done in the cloud. The key advantages of this mechanism that may not be available in server-based approaches are:

- When serverless functions are not being used (during some local computations in an algorithm), since there are no servers running, users do not pay for the resources they do not use.
- The serverless approach has the ease of running the overall algorithm in your own machine with only computationally-heavy parts running on the cloud and returning their outputs to your local machine for further manipulation.
- It is possible to request different numbers of serverless functions in different stages of the algorithm. This might be cost-effective when running algorithms with varying computational loads.

Along with these advantages, serverless computing comes with certain limitations for resources such as RAM and lifetime. For instance, for AWS Lambda, each function can have a maximum of 3 GB of RAM and has a lifetime up to 15 minutes. A mechanism based on serverless computing must take these limitations into account.

One problem that serverless and server-based computing platforms have in common is that a portion of the functions (or workers) might finish their task later than the others (these slower workers are referred to as stragglers), and this may cause the overall output to be delayed if no precaution is taken. To overcome the issue of straggling functions, we use the idea of inserting redundancy to computations using codes. This idea has been explored in the literature by many works such as [9], [2], [14] for dealing with stragglers encountered in distributed computation. Codes not only help speed up computations by making it possible to compute the desired output without waiting for the outputs of the straggling workers, but also provide resilience against crashes and timeouts, leading to a more reliable mechanism.

A straggler-resilient scheme designed for a serverless computing platform needs to be scalable as we wish to allow the number of functions to vary greatly. Two things become particularly important if the number of workers is as high as hundreds or thousands. The first one is that encoding and decoding must be low complexity. The second one is that one must be careful with the numerical round-off errors if the inputs are not from a finite field, but instead are full-precision real numbers. To clarify this point, when the inputs are real-valued, encoding and decoding operations introduce round-off errors. Polar codes show superiority over many codes in terms of both of these aspects. They have low encoding and decoding complexity and both encoding and decoding involve a small number of subtraction and addition operations (no multiplications involved in encoding or decoding). In addition, they achieve capacity, and the importance of that fact in coded computation is that the number of worker outputs needed for decoding is asymptotically optimal.

A. Server-Based vs Serverless Computing

We note an important difference between using serverless and server-based computing that helps highlight the usefulness of polar codes. In server-based computing, one needs to use much fewer machines than the number of functions one would need in serverless computing to achieve the same amount of computing. The reason for this is the limited resources each function can have in serverless computing. This is an important design criterion that needs to be addressed as it necessitates efficient encoding and decoding algorithms for the code we are using. For instance, for a distributed server-based system with $N = 8$ machines, using maximum distance separable (MDS) codes with decoding complexity as high as $O(N^3)$ could still be acceptable. This becomes unacceptable when we switch to a serverless system as the number of functions that achieve the same amount of computations could go up as high as a few hundreds. In this case, it becomes necessary to use codes with fast decoders such as polar codes.

Even though polar codes do not have the MDS code properties, as the block-length of the code increases (i.e. more functions are used), the performance gap between polar codes and MDS codes closes. We further discuss this point in the Numerical Results section.

B. Related Work

Coded matrix multiplication has been introduced in [9] for speeding up distributed matrix-vector multiplication in server-based computing platforms. In [9], it was shown that it is possible to speed up distributed matrix multiplication by using codes (MDS codes in particular). MDS codes however have the disadvantage of having high encoding and decoding complexity, which could be restricting in setups with large number of workers. The work in [2] attacks at this problem presenting a coded computation scheme based on d-dimensional product codes. [14] presents a scheme referred to as polynomial codes for coded matrix multiplication with input matrices from a large finite field. This approach might require quantization for real-valued inputs which could introduce additional numerical issues. [5] and [15] are other works investigating coded matrix multiplication and provide analysis on the optimal number of worker outputs required. In addition to the coding theoretic approaches, [7] offers an *approximate* straggler-resilient matrix multiplication scheme where the ideas of sketching and straggler-resilient distributed matrix multiplication are brought together.

Using Luby Transform (LT) codes, a type of rateless fountain codes, in coded computation has been recently proposed in [11] and [10]. The proposed scheme in [10] divides the overall task into smaller tasks (each task is a multiplication of a row of A with x) for better load-balancing. The work [11] proposes the use of inactivation decoding and the work [10] uses peeling decoder. Peeling decoder has a computational complexity of $O(N \log N)$ (same complexity as the decoder we propose), however its performance is not satisfactory if the number of inputs symbols is not very large. Inactivation decoder performs better than the peeling decoder, however, it is not as fast as the peeling decoder.

Among the recent work on serverless computing for machine learning training is [3] where serverless machine learning is discussed in detail, and challenges and possible solutions on serverless machine training are provided. Authors in [6] consider an architecture with multiple master nodes and state that for small neural network models, serverless computing helps speed up hyperparameter tuning. Similarly, the work in [13] shows via experiments that their prototype on AWS Lambda can reduce model training time greatly.

C. Main Contributions

This work investigates the use of serverless computing for large-scale computations with a solution to the issue of stragglers based on polar codes. We identify that polar codes are a natural choice for coded computation with serverless computing due to their low complexity encoding and decoding. Furthermore, we propose a sequential decoder for polar codes designed to work with full-precision data for erasure channels. The proposed decoder is different from successive cancellation (SC) decoder given in [1], which is for discrete data and is based on estimation using likelihoods. We also analytically justify the use of the polar code kernel that we used in this work when the inputs and outputs are not from a finite field but are full-precision real-valued data. Next, we discuss the polarization of distributions of worker run times and provide some numerical results to that end. This provides some important insights towards understanding polarization phenomenon in a setting other than communication.

Given the work of [9], it is not very surprising that any linear code such as polar codes could also be used in coded computation. The main novelty of this work lies in the design of a mechanism based on serverless computing that is easy to use and can handle large-scale data. We adopt coded computation as the tool to make the mechanism more resilient and reliable, and polar codes are our choice of code to ensure scalability.

Differently from other erasure codes, polar codes offer a very interesting interpretation in terms of run times of machines due to the polarization phenomenon. By bringing together a number of machines in polar code construction, we achieve virtually transformed machines with polarized run times. To clarify, the polar coding transformation polarizes the run times of the workers, that is, as a result of the transformation, we will obtain faster and slower transformed machines. This interpretation is equivalent to channels polarizing when polar codes are used for channel coding in communications [1]. If we set the rate to $1 - \epsilon$, then we choose the best $N(1 - \epsilon)$ machines out of the N machines as the data machines whereas the remaining machines are frozen (i.e. we send in zero matrices for the slow machines).

II. Coded Computation with Polar Coding

In this section we describe our proposed polar coding method for matrix-vector multiplication. Later in coded matrix multiplication section, we talk about the extension of this method to matrix multiplication (both matrices are encoded instead of one). We would like to note that the method described in this section is applicable for both server-based and serverless computing platforms. Hence, we use the terms *worker*, *serverless function*, *compute node* to mean the same thing, which is a single computing block of whatever distributed platform we are using. The term master node refers to the machine that we aggregate the results in.

A. System Model

We are interested in speeding up the computation of $A \times x$ where $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^{n \times r}$ using the model in Fig. 1. The master node encodes the A matrix and communicates the encoded matrix chunks to workers. Compute nodes then perform their assigned matrix multiplication. After compute nodes start performing their assigned tasks, the master node starts listening for compute node outputs. Whenever a decodable set of outputs is detected, the master node downloads the available node outputs and performs decoding, using the sequential decoder described in subsection II.E.

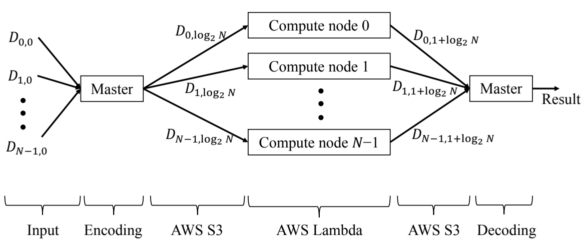


Fig. 1. System model.

Note that this approach assumes x fits in the memory of a compute node. In coded computation literature, this is referred to as large-scale matrix-vector multiplication, different from matrix-matrix multiplication where the second matrix is also partitioned, which discuss later in the text.

B. Code Construction

Consider the polar coding representations in Fig. 2 and 3 for $N = 2$ and 4 nodes. The channels W are erasure channels because of the assumption that if the output of a compute node is available, then it is assumed to be correct. If the output of a node is not available (straggler node), then it is considered an erasure. Let D_{ij} denote the data for node i in the j'th level[1]. Further, let us denote the erasure probability of each channel by ϵ, and assume that the channels are independent. It is straightforward to calculate the polarized erasure probabilities for the transformed channels, which we skip (we refer the readers to [1] for more on polar codes). Based on the erasure probabilities of the transformed channels, we select the best channels for data, and freeze the rest (i.e., set to zero matrices).

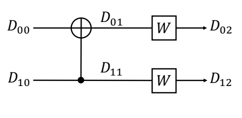


Fig. 2. Polar coding for $N = 2$.

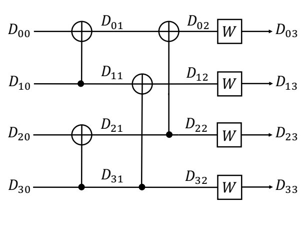


Fig. 3. Polar coding for $N = 4$.

A possible disadvantage of the proposed scheme is that the number of compute nodes has to be a power of 2. We identify two possible solutions to overcome this issue. The first one is to consider using a kernel of different size instead of 2. The second solution is to have multiple code constructions. For instance, let us assume that we wish to use 20 compute nodes and we choose $\epsilon = \frac{1}{4}$. We can divide the overall task into 2 constructions. We would compute the first task using 16 compute nodes (12 data, 4 frozen channels), and the second task using 4 compute nodes (3 data, 1 frozen channels), where the first task is to multiply the first $\frac{4}{5}m$ rows of A by x, and the second is to multiply the last $\frac{1}{5}m$ rows of A by x.

C. Encoding Algorithm

After computing the erasure probabilities for the transformed channels, We choose the $d = N(1-\epsilon)$ channels with the lowest erasure probabilities as data channels. The remaining $N\epsilon$ channels are frozen. The inputs for the data channels are the row partitions of matrix A. That is, we partition A such that each matrix chunk has $\frac{m}{d}$ rows: $A = [A_1^T, A_2^T, \ldots A_d^T]^T$ where $A_i \in \mathbb{R}^{\frac{m}{d} \times n}$. For example, for $N = 4$ and $\epsilon = 0.5$, the erasure probabilities of the transform channels are calculated to be $[0.938, 0.563, 0.438, 0.063]$. It follows that we freeze the first two channels, and the last two channels are the data channels. This means that in Fig. 3, we set $D_{00} = D_{10} = 0^{\frac{m}{2} \times n}$ and $D_{20} = A_1$, $D_{30} = A_2$.

We encode the input using the standard polar coding circuit (given for $N = 4$ in Fig. 3). Note that instead of the XOR operation like in polar coding for communication, what we have in this work is addition over real numbers. The way encoding is done is that we compute the data for each level from left to right starting from the leftmost level. There are $\log_2 N + 1$ levels in total and N nodes in every level. Hence, the encoding complexity is $O(N \log N)$.

D. Channel

Channels in the communication context correspond to compute nodes in the coded computation framework. The i'th node computes the matrix multiplication $D_{i,\log_2 N} \times x$. When a decodable set of outputs is detected, the master node moves on to decoding the available outputs. At that time, all the delayed nodes (stragglers) are considered to be erased.

E. Decoding Algorithm

The decoding algorithm, whose pseudo-code is given in Alg. 1, does not require quantization of data as it is tailored to work with full-precision data. Note that the notation $I_{D_{ij}} \in \{0, 1\}$ indicates whether the data for node i in level j (namely, D_{ij}) is known. In the first part of the decoding algorithm, we continuously check for decodability as compute node outputs become available. When the available outputs become decodable, we move on to the second part where decoding takes place at the master node given the available compute node outputs. The second part makes calls to the function *decodeRecursive*, given in Alg. 2.

III. Analysis

Note that both encoding and decoding algorithms operate only on $N = 2$ blocks which correspond to the kernel $F_2 = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$. Theorem 1 justifies the use of F_2 for real-valued data

[1]Note that during encoding, the node values D_{ij} represent the data before it is multiplied by x, and during decoding, D_{ij}s represent the data after the multiplication by x.

[2]checkDecodability is a function that checks whether it is possible to decode a given sequence of indicators (identifying availability of outputs). This works the same way as part II of Algorithm 1, but differs in that it does not deal with data, but instead binary indicators.

Algorithm 1: Decoding algorithm.

```
Input: Indices of the frozen channels
Result: $y = A \times x$                                  ▷ Part I
Initialize $I_{D_{:,0}} = [I_{D_{0,0}}, I_{D_{1,0}}, \dots, I_{D_{N\ 1,0}}] = [0, \dots, 0]$
while $I_{D_{:,0}}$ not decodable do
    update $I_{D_{:,0}}$
    checkDecodability($I_{D_{:,0}}$)[2]
Initialize an empty list $y$                              ▷ Part II
for $i \leftarrow 0$ to $N-1$ do
    $D_{i,0}$ = decodeRecursive($i$, 0)
    if node $i$ is a data node then
        $y = [y; D_{i,0}]$
    if $i \bmod 2 = 1$ then                               ▷ forward prop
        for $j \leftarrow 0$ to $\log_2 N$ do
            for $l \leftarrow 0$ to $i$ do
                compute $D_{lj}$ if unknown
```

Algorithm 2: decodeRecursive(i, j)

```
Input: Node $i \in [0, N-1]$, level $j \in [0, \log_2 N]$
Result: $I_{D_{ij}}$ and modifies $D$
if $j = \log_2 N$ then return $I_{D_{i,\log_2 N}}$        ▷ base case 1
if $I_{D_{ij}} = 1$ then return 1                          ▷ base case 2
$I_{D_{i,(j+1)}}$ = decodeRecursive($i, j+1$)
$I_{D_{pair(i),(j+1)}}$ = decodeRecursive($pair(i), j+1$)
if $i$ is upper node then
    if $I_{D_{i,(j+1)}}$ AND $I_{D_{pair(i),(j+1)}} = 1$ then
        compute $D_{ij}$
        return 1
else
    if $I_{D_{i,(j+1)}}$ OR $I_{D_{pair(i),(j+1)}}$ = 1 then
        compute $D_{ij}$
        return 1
return 0
```

among all possible 2×2 kernels. We first give a definition for a *polarizing kernel*, and then give a lemma which will later be used in proving Theorem 1.

Definition 1 (Polarizing kernel): Let f be a function satisfying the linearity property $f(au_1 + bu_2) = af(u_1) + bf(u_2)$ where $a, b \in \mathbb{R}$ and assume that there is an algorithm to compute f that takes a certain amount of time to run with its run time distributed randomly. Let K denote a 2×2 kernel and $\left[\begin{smallmatrix} v_1 \\ v_2 \end{smallmatrix}\right] = K \times \left[\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix}\right]$. Assume that we input v_1 and v_2 to two i.i.d. instances of the same algorithm for f. Further, let T_1, T_2 be random variables denoting the run times for computing $f(v_1)$, $f(v_2)$, respectively. We are interested in computing $f(u_1)$, $f(u_2)$ in this order. If the time required to compute $f(u_1)$ is $\max(T_1, T_2)$ and the time required to compute $f(u_2)$ given the value of $f(u_1)$ is $\min(T_1, T_2)$, then we say K is a polarizing kernel.

Lemma 1: A kernel $K \in \mathbb{R}^{2\times 2}$ is a polarizing kernel if and only if the following conditions are both satisfied: 1) Both elements in the second column of K are nonzero, 2) K is invertible.

Proof: We first prove that if K is a polarizing kernel, then it satisfies both of the given conditions. Let us assume an f function satisfying the linearity property given in Definition 1. Since f holds the linearity property, we can write

$$\begin{bmatrix} f(v_1) \\ f(v_2) \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \times \begin{bmatrix} f(u_1) \\ f(u_2) \end{bmatrix}. \qquad (1)$$

Computing $f(u_2)$ given the value of $f(u_1)$ in time $\min(T_1, T_2)$ means that $f(u_2)$ can be computed using $f(u_1)$ and either one of $f(v_1)$, $f(v_2)$ (whichever is computed earlier). This implies that we must be able to recover $f(u_2)$ using one of the following two equations

$$K_{12} \times f(u_2) = f(v_1) - K_{11} f(u_1) \qquad (2)$$

$$K_{22} \times f(u_2) = f(v_2) - K_{21} f(u_1). \qquad (3)$$

We use (2) if $f(v_1)$ is known, and (3) if $f(v_2)$ is known. This implies that both K_{12} and K_{22} need to be nonzero. Furthermore, to be able to compute $f(u_1)$ in time $\max(T_1, T_2)$ means that it is possible to find $f(u_1)$ using both $f(v_1)$ and $f(v_2)$ (note that we do not assume we know the value of $f(u_2)$). There are two scenarios where this is possible: Either at least one row of K must have its first element as nonzero and its second element as zero, or K must be invertible. Since we already found out that K_{12} and K_{22} are both nonzero, we are left with one scenario, which is that K must be invertible.

We proceed to prove the other direction of the 'if and only if' statement, which states that if a kernel $K \in \mathbb{R}^{2\times 2}$ satisfies the given two conditions, then it is a polarizing kernel. We start by assuming an invertible $K \in \mathbb{R}^{2\times 2}$ with both elements in its second column nonzero. Since K is invertible, we can uniquely determine $f(u_1)$ when both $f(v_1)$ and $f(v_2)$ are available, which occurs at time $\max(T_1, T_2)$. Furthermore, assume we know the value of $f(u_1)$. At time $\min(T_1, T_2)$, we will also know one of $f(v_1)$, $f(v_2)$, whichever is computed earlier. Knowing $f(u_1)$, and any one of $f(v_1)$, $f(v_2)$, we can determine $f(u_2)$ using the suitable one of the equations (2), (3) because K_{12} and K_{22} are both assumed to be nonzero. Hence this completes the proof that a kernel K satisfying the given two conditions is a polarizing kernel. ■

Theorem 1: Of all possible 2×2 polarizing kernels, the kernel $F_2 = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ results in the fewest number of computations for encoding real-valued data.

Proof: By Lemma 1, we know that for K to be a polarizing kernel, it must be invertible and must have both K_{12} and K_{22} as nonzero. For a 2×2 matrix to be invertible with both second column elements as nonzero, at least one of the elements in the first column must also be nonzero. We now know that K_{12}, K_{22} and at least one of K_{11}, K_{21} must be nonzero in a polarizing kernel K. It is easy to see that having all four elements of K as nonzero leads to more computations than having only three elements of K as nonzero. Hence, we must choose either K_{11} or K_{21} to be zero (it does not matter which one). It is possible to avoid any multiplications by selecting the nonzero elements of K as ones. Hence, both $K = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ and $K = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ are polarizing kernels and lead to the same amount of computations, which is a single addition. This amount of computations is the minimum possible as otherwise K will not satisfy the condition that a polarizing kernel must have at least 3 nonzero elements. ■

IV. Discussion on Fast Decoders for MDS Codes

Most works in coded computation literature employ MDS codes as their way of inserting redundancy into computations.

In the case where one wishes to work with full-precision data and use Reed-Solomon codes, the decoding requires solving a linear system. Hence decoding becomes expensive (using Gaussian elimination it takes $O(n^3)$ operations) and gets unstable for systems with high number of workers since we are solving a Vandermonde based linear system.

There are many works that restrict their schemes to working with values from a finite field of some size q. In that case, it is possible to use fast decoding algorithms which are based on fast algorithms for polynomial interpolation. One such decoding algorithm is given in [12], which provides $O(n \log n)$ encoding and decoding algorithms for Reed-Solomon erasure codes based on Fermat Number Transform (FNT). The complexity for the encoder is the same as taking a single FNT transform and for the decoder, it is equal to taking 8 FNT transforms. To compare with polar codes, polar codes require $n \log n$ operations for both encoding and decoding. Another work where a fast erasure decoder for Reed-Solomon codes is presented is [4] which presents a decoder that works in time $O(n \log^2 n)$.

There exist many other fast decoding algorithms for RS codes with complexities as low as $O(n \log n)$. However, the decoding process in these algorithms usually requires taking a fast transform (e.g. FNT) many times and are limited to finite fields. Often these fast decoding algorithms have big hidden constants in their complexity and hence quadratic time decoding algorithms are sometimes preferred over them. Polar codes, on the other hand, provide very straightforward and computationally inexpensive encoding and decoding algorithms. One of our contributions is the design of an efficient decoder for polar codes tailored for the erasure channel that can decode full-precision data.

V. Partial Construction

In this section, we introduce a new idea that we refer to as *partial construction*. Let us assume we are interested in computing Ax where we only encode A and not x. If the data matrix A is very large, it might be challenging to encode it. A way around having to encode a large A is to consider partial code constructions, that is, for $A = [A_1^T, ..., A_p^T]^T$, we encode each submatrix A_i separately. It follows that decoding the outputs of the construction for the submatrix A_i will give us $A_i x$. This results in a weaker straggler resilience, however, we get a trade-off between the computational load of encoding and straggler resilience. Partial construction also decreases the computations required for decoding since instead of decoding a code with N outputs (of complexity $O(N \log N)$), now we need to decode p codes with $\frac{N}{p}$ outputs, which is of total complexity $O(N \log(\frac{N}{p}))$.

In addition, partial construction makes it possible to parallel compute both encoding and decoding. Each code construction can be encoded and decoded independently from the rest of the constructions. Partial construction idea can also be applied to coded computation schemes based on other codes. For instance, one scenario where this idea is useful is when one is interested in using RS codes with full-precision data. Given that for large N values, using RS codes with full-precision data becomes impossible, one can construct many smaller size codes. When the code size is small enough, a Vandermonde-based linear system can be painlessly solved.

In-Memory Encoding: Another benefit of the partial construction idea is that for constructions of sizes small enough, encoding can be performed in the memory of the workers after reading the necessary data. This results in a straggler-resilient scheme without doing any pre-computing to encode the entire dataset. In-memory encoding could be useful for problems where the data matrix A also is also changing over time because it might be too expensive to encode the entire dataset A every time it changes.

VI. Coded Matrix Multiplication

In this section, we provide an extension to our proposed method to accommodate coding of both A and B for computing the matrix multiplication AB (instead of coding only A). This can be thought of as a two-dimensional extension of our method. Let $A = [A_1^T, ..., A_{d_1}^T]^T, B = [B_1, ..., B_{d_2}]$. Let us denote zero matrix padded version of A by $\tilde{A} = [\tilde{A}_1^T, ..., \tilde{A}_{N_1}^T]^T$ such that $\tilde{A}_i = 0$ if i is a frozen channel index and $\tilde{A}_i = A_j$ if i is a data channel index with j the appropriate index. Similarly, we define $\tilde{B} = [\tilde{B}_1, ..., \tilde{B}_{N_2}]$ such that $\tilde{B}_i = 0$ if i is a frozen channel index and $\tilde{B}_i = B_j$ if i is a data channel index with j the appropriate index. Encoding on $\tilde{A}$ can be represented as $G_{N_1}\tilde{A}$ where G_{N_1} is the N_1 dimensional generator matrix and acts on submatrices $\tilde{A}_i$. Similarly, encoding on $\tilde{B}$ would be $\tilde{B}G_{N_2}$.

Encoding A and B gives us N_1 submatrices $(G_{N_1}\tilde{A})_i$ due to A and N_2 submatrices $(\tilde{B}G_{N_2})_j$ due to B. We multiply the encoded matrices using $N_1 N_2$ workers with the (i, j)'th worker computing the multiplication $(G_{N_1}\tilde{A})_i(\tilde{B}G_{N_2})_j$. So, the worker outputs will be of the form:

$$P = \begin{bmatrix} (G_{N_1}\tilde{A})_1(\tilde{B}G_{N_2})_1 & \dots & (G_{N_1}\tilde{A})_1(\tilde{B}G_{N_2})_{N_2} \\ \vdots & \ddots & \\ (G_{N_1}\tilde{A})_{N_1}(\tilde{B}G_{N_2})_1 & \dots & (G_{N_1}\tilde{A})_{N_1}(\tilde{B}G_{N_2})_{N_2} \end{bmatrix} \tag{4}$$

Note that for fixed j, the worker outputs are:

$$\begin{bmatrix} (G_{N_1}\tilde{A})_1(\tilde{B}G_{N_2})_j \\ \vdots \\ (G_{N_1}\tilde{A})_{N_1}(\tilde{B}G_{N_2})_j \end{bmatrix} \tag{5}$$

For fixed j, the outputs are linear in $(\tilde{B}G_{N_2})_j$, hence, it is possible to decode these outputs using the decoder we have for the 1D case. Similarly, for fixed i, the outputs are:

$$\begin{bmatrix} (G_{N_1}\tilde{A})_i(\tilde{B}G_{N_2})_1 & \dots & (G_{N_1}\tilde{A})_i(\tilde{B}G_{N_2})_{N_2} \end{bmatrix} \tag{6}$$

For fixed i, the outputs are linear in $(G_{N_1}\tilde{A})_i$. It follows that the 1D decoding algorithm can be used for decoding the outputs. Based on these observations, we designed a polar decoder for the 2D case whose pseudo-code is given in Alg. 3. The 2D decoding algorithm makes calls to the 1D encoding and decoding algorithms many times to fill in the missing entries of the encoded matrix P defined in (4). When all missing entries of P are computed, first all rows and then all columns of P are decoded and finally, the frozen entries are removed to obtain the multiplication AB.

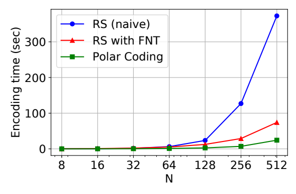
Algorithm 3: 2D decoding algorithm.

```
Input: the worker output matrix $P$
Result: $y = A \times B$
while $P$ has missing entries do
    for $i \leftarrow 0$ to $N_1 - 1$ do        ▷ loop over rows
        if $P[i,:]$ has missing entries and is decodable
          then
            decode $P[i,:]$ using Alg. 1
            forward prop to fill in $P[i,:]$
    for $j \leftarrow 0$ to $N_2 - 1$ do ▷ loop over columns
        if $P[:,j]$ has missing entries and is decodable
          then
            decode $P[:,j]$ using Alg. 1
            forward prop to fill in $P[:,j]$
decode all rows and then all columns of $P$
return entries of $P$ (ignoring the frozen entries)
```

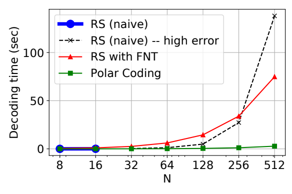
VII. Numerical Results

A. Encoding and Decoding Speed Comparison

Fig. 4 shows the time encoding and decoding algorithms take as a function of the number of workers N for Reed-Solomon and polar codes. For Reed-Solomon codes, we implemented two encoders and decoders. The first approach is the naive approach where encoding is done using matrix multiplication ($O(N^2)$) and decoding is done by solving a linear system ($O(N^3)$), hence the naive encoder and decoder can support full-precision data. The second approach is the fast implementation for both encoding and decoding (of complexities $O(N \log N)$ and $O(N \log^2 N)$, respectively). The fast implementation is based on Fermat Number Transform (FNT), hence only supports finite field data. In obtaining the plots in Fig. 4, we always used 0.5 as the rate and performed the computation Ax where A is $(100N \times 5000)$-dimensional and x is (5000×1000)-dimensional. The line in Fig. 4(b) with markers 'x' and dashed lines indicates that the error due to the decoder is unacceptably high (this can happen because we used full-precision data for the naive RS decoder).



(a) Encoding



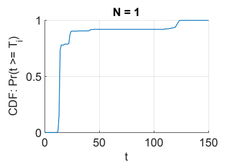
(b) Decoding

Fig. 4. Comparison of encoding and decoding speeds for RS and polar codes. Note that the horizontal axis is in log-scale.

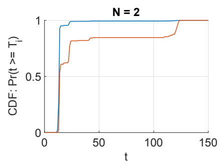
Fig. 4 illustrates that polar codes take much less time for encoding and especially decoding compared to Reed-Solomon codes. This is because of the constants hidden in the complexities of fast decoders for Reed-Solomon decoders which is not the case for polar codes. We note that it might be more advantageous to use RS codes for small N values because they have the MDS properties and can encode and decode fast enough for small N. However, in serverless computing where each function has limited resources and hence using large N values is usually the case, we need faster encoding and decoding algorithms. Considering the comparison in Fig. 4, polar codes are more suitable to use with serverless computing.

B. Polarized Computation Times

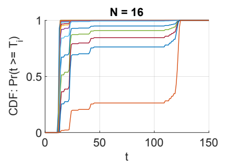
Fig. 5 illustrates the polarization of the empirical CDFs of computation times. Fig. 5(a) shows the empirical CDF of the computation times of AWS Lambda functions, obtained by timing 500 AWS Lambda functions running the same Python program. The other plots in Fig. 5 are generated assuming that there are N functions with i.i.d. run time distributions (the CDF of which is in Fig. 5 (a)), and show the resulting transformed CDFs. Note that the empirical CDFs move away from each other (i.e. polarize) as N is increased and we obtain *better* and *worse* run time distributions. Note that the freezing operation in the code construction corresponds to not using workers with worse run times.



(a) $N = 1$



(b) $N = 2$



(c) $N = 16$

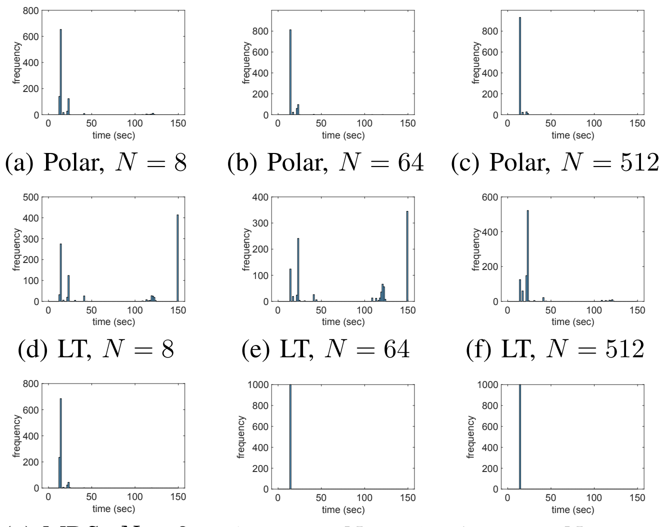
Fig. 5. Polarization of empirical CDFs.

C. Empirical Distribution of Decodability Time

We refer to the time instance where the available outputs become decodable for the first time as *decodability time*. Fig. 6 shows the histograms of the decodability time for different values of N for polar, LT, and MDS codes, respectively. These histograms were obtained by sampling i.i.d. worker run times with replacement from the input distribution whose CDF is plotted in Fig. 5(a) and by repeating this 1000 times. Further, $\epsilon = 0.375$ was used as the erasure probability. We observe that as N increases, the distributions converge to the dirac delta function, showing that for large N values, the decodability time becomes deterministic.

Plots in Fig. 6(d,e,f) are the decodability time histograms for LT codes with peeling decoder. The degree distribution is the robust soliton distribution as suggested in [10]. We see that polar codes achieve better decodability times than LT codes. Plots in Fig. 6(g,h,i) on the other hand show that MDS codes perform better than polar codes in terms of decodability time, which is expected. When considering this result, one should keep in mind that for large N, MDS codes take much longer times to encode and decode compared to polar codes as we discussed previously. In addition, we see that for large N values, the gap between the decodability time performances closes.

Fig. 7 shows decodability time histograms for computing the matrix multiplication AB for polar (using the 2D decoder in Alg. 3) and MDS codes. The rates for encoding A and B are both $\frac{3}{4}$ and we have an overall rate of $\frac{9}{16}$. The decodability time for polar codes is the first time instance in which the decoder in Alg. 3 is able to decode the available set of outputs. The decodability time for MDS codes is

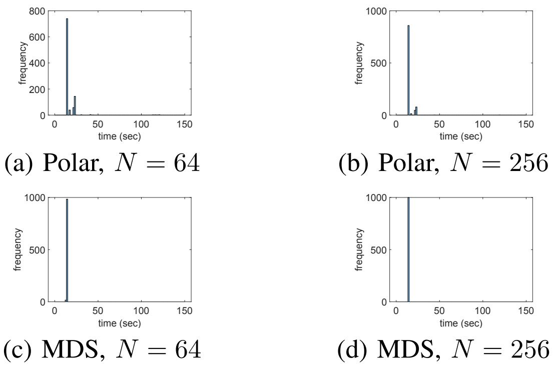

(a) Polar, $N = 8$ (b) Polar, $N = 64$ (c) Polar, $N = 512$

(d) LT, $N = 8$ (e) LT, $N = 64$ (f) LT, $N = 512$

(g) MDS, $N = 8$ (h) MDS, $N = 64$ (i) MDS, $N = 512$

Fig. 6. Histograms of decodability time for polar, LT, and MDS codes.

whenever $\frac{9N}{16}$ of the outputs is available. Fig. 7 illustrates that the decodability time for the polar codes is only slightly worse than for MDS codes. Note that the decodability time for MDS codes is optimal since the recovery threshold is optimal for MDS codes when A is partitioned row-wise and B is partitioned column-wise when computing AB [14]. For $N = 256$, Fig. 7 shows that the gap between the decodability times of polar and MDS codes is negligibly small. Hence, as the number of functions is increased, the decodability time becomes no longer an important disadvantage of using polar codes instead of MDS codes.


(a) Polar, $N = 64$ (b) Polar, $N = 256$

(c) MDS, $N = 64$ (d) MDS, $N = 256$

Fig. 7. Histograms of decodability time for matrix multiplication.

D. Coded Computation on AWS Lambda

We have implemented the proposed coded computation scheme. The implementation is written in Python. The compute nodes are AWS Lambda functions each with a memory of 1536 MB and a timeout limit of 300 sec. We have utilized Pywren [8] for running code in AWS Lambda. The master node is a Macbook Pro with 8 GB of RAM. Fig. 8 shows the timings of serverless functions, and of the decoding process, where we compute Ax with $A \in \mathbb{R}^{38400\times 3000000}$ and $x \in \mathbb{R}^{3000000\times 20}$ using $N = 512$, and taking $\epsilon = 0.25$. The vertical axis values correspond to different functions. Encoding process is not included in the plot, as it is usually the case for many applications that we multiply the same A matrix with different x's. Hence for an application requiring iterative matrix multiplications with the same A but different x's, it is sufficient to encode A once, but decoding has to be repeated for different x's. Therefore, the overall run time is dominated by the matrix multiplications and decoding.

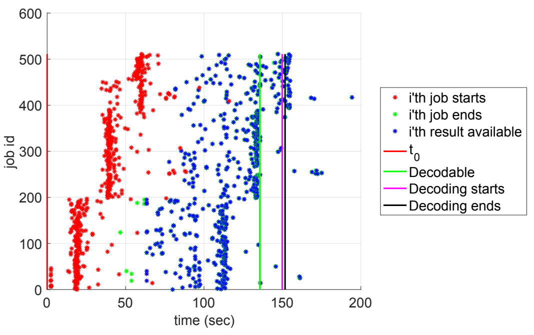


Fig. 8. Job output times and decoding times for $N = 512$ on AWS Lambda.

For the experiment shown in Fig. 8, each serverless function instance performs a matrix multiplication with dimensions (100×3000000) and (3000000×20). A matrix with dimensions (100×3000000) requires a memory of 2.3 GB. It is not feasible to load a matrix of this size into the memory at once; hence, each serverless function loads $1/10$'th of its corresponding matrix at a time. After computing $1/10$'th of its overall computation, it moves on to perform the next $1/10$ of its computation. In Fig. 8, the red, green, and blue stars represent the starting, ending, and output availability times for each function, respectively. Green (vertical) line is the first time the available outputs become decodable. After the available outputs become decodable, we cannot start decoding right away; we have to wait for their download (to the master node) to be finished, and this is shown by the magenta line. Magenta line also indicates the beginning of the decoding process, and lastly the black line indicates that decoding is finished.

We observe that there are straggling AWS Lambda functions that finish after decoding is over, showing the advantage of our coded computation scheme over an uncoded scheme where we would have to wait for all AWS Lambda functions to return their outputs. The slowest serverless function seems to have finished around $t = 200$, whereas the result is available around $t = 150$ for the proposed coded computation.

E. Gradient Descent on a Least Squares Problem

We used the proposed method in solving a least squares problem via gradient descent using AWS Lambda. The problem that we focus on in this subsection is the standard least squares problem:

$$\text{minimize } ||Ax - y||_2^2 \tag{7}$$

where $x \in \mathbb{R}^{n\times r}$ is not necessarily a vector; it could be a matrix. Let $x^* \in \mathbb{R}^{n\times r}$ be the optimal solution to (7), and let x_i^* represent the i'th column of x^*. Note that if x is a matrix, then the problem (7) is equivalent to solving r different least-squares problems with the same A matrix: minimize $||Ax_i -$

$y_i||_2^2$. The solution to this problem is the same as the i'th column of x^*, that is, x_i^*. When r is large (and if we assume that A fits in the memory of each worker), one way to solve (7) in a distributed setting is by assigning the problem of minimizing $||Ax_i - y_i||_2^2$ to worker i. This solution however would suffer from possible straggling serverless function, and assumes that A is small enough to fit in the memory of a serverless function. Another solution, the one we experiment with in this subsection, is to use gradient descent as in this case we are no longer restricted by A having to fit in the memory and our proposed method can be used to provide resilience towards stragglers for this solution.

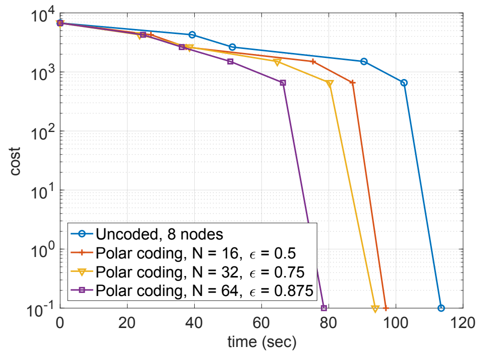


Fig. 9. Cost vs time for the gradient descent example.

The gradient descent update rule for this problem is:

$$x_{t+1} = x_t - \mu(A^T A x_t - A^T y), \tag{8}$$

where μ is the step size (or learning rate), and the subscript t specifies the iteration number.

We applied our proposed coded matrix multiplication scheme to solve a least squares problem (7) using gradient descent where $A \in \mathbb{R}^{20000\times4800}$, and $x \in \mathbb{R}^{4800\times1000}$. Fig. 9 shows the downward-shifted cost (i.e., l_2-norm of the residuals $||Ax_t - y||_2$) as a function of time under different schemes. Before starting to update the gradients, we first compute and encode $A^T A$ offline. Then, in each iteration of the gradient descent, the master node decodes the downloaded outputs, updates x_t, sends the updated x_t to AWS S3 and initializes the computation $A^T A x_t$. In the case of uncoded computation, we simply divide the multiplication task among $N(1-\epsilon)$ serverless functions, and whenever all of the $N(1-\epsilon)$ functions finish their computations, the outputs are downloaded to the master node, and there is no decoding. Then, master node computes and sends the updated x_t, and initializes the next iteration.

Note that in a given iteration, while computation with polar coding with rate $(1-\epsilon)$ waits for the first decodable set of outputs out of N outputs, uncoded computation waits for all $N(1-\epsilon)$ nodes to finish computation. With this in mind, it follows that coded computation results in a trade-off between price and time, that is, by paying more, we can achieve a faster convergence time, as illustrated in Fig. 9. Note that in coded computation we pay for N serverless functions, and in uncoded computation we pay for $N(1-\epsilon)$ serverless functions. Using ϵ as a tuning parameter for redundancy, we achieve different convergence times.

VIII. Conclusion

We have proposed to use polar codes for distributed matrix multiplication in serverless computing, and discussed that the properties of polar codes are a good fit for straggler-resilient serverless computing. The proposed mechanism addresses the limitations of serverless computing. We considered a centralized model with a master node and a large number of serverless functions. We implemented our proposed method using AWS Lambda. We presented a sequential decoder algorithm that can encode and decode full-precision data so that our proposed framework can be used for computations on full-precision data. We discussed how to extend the proposed method to the matrix-matrix multiplication case where we code both matrices being multiplied. We have identified and illustrated with a numerical example a trade-off between the computation price and convergence time for the gradient descent algorithm applied to a least-squares problem.

Acknowledgements

This work was supported in part by the National Science Foundation under grant IIS-1838179.

References

[1] E. Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on Information Theory*, 55:3051–3073, 2009.

[2] T. Baharav, K. Lee, O. Ocal, and K. Ramchandran. Straggler-proofing massive-scale distributed matrix multiplication with d-dimensional product codes. *IEEE International Symposium on Information Theory (ISIT)*, pages 1993–1997, 2018.

[3] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, , and Randy Katz. A case for serverless machine learning. *Workshop on Systems for ML and Open Source Software at NeurIPS 2018*, 2018.

[4] Frédéric Didier. Efficient erasure decoding of reed-solomon codes. *CoRR*, abs/0901.1886, 2009.

[5] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. R. Cadambe, and P. Grover. On the optimal recovery threshold of coded matrix multiplication. *CoRR*, abs/1801.10292, 2018.

[6] L. Feng, P. Kudva, D. Da Silva, and J. Hu. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341, July 2018.

[7] V. Gupta, S. Wang, T. A. Courtade, and K. Ramchandran. Oversketch: Approximate matrix multiplication for the cloud. *CoRR*, abs/1811.02653, 2018.

[8] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. *CoRR*, abs/1702.04024, 2017.

[9] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, March 2018.

[10] A. Mallick, M. Chaudhari, and G. Joshi. Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication. *CoRR*, abs/1804.10331, 2018.

[11] A. Severinson, A. G. i Amat, and E. Rosnes. Block-diagonal and lt codes for distributed computing with straggling servers. *IEEE Transactions on Communications*, 2018.

[12] A. Soro and J. Lacan. Fnt-based reed-solomon erasure codes. In *7th IEEE Consumer Comm. and Networking Conf.*, pages 1–5, 2010.

[13] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, April 2019.

[14] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr. Polynomial codes: An optimal design for high-dimensional coded matrix multiplication. *Adv. in Neural Info. Proc. Systems (NIPS) 30*, pages 4406–4416, 2017.

[15] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr. Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding. *CoRR*, abs/1801.07487, 2018.