Randomized Polar Codes for Anytime Distributed Machine Learning[*]

Burak Bartan[†] and Mert Pilanci[‡]

Abstract

We present a novel distributed computing framework that is robust to slow compute nodes, and is capable of both approximate and exact computation of linear operations. The proposed mechanism integrates the concepts of randomized sketching and polar codes in the context of coded computation. We propose a sequential decoding algorithm designed to handle real valued data while maintaining low computational complexity for recovery. Additionally, we provide an anytime estimator that can generate provably accurate estimates even when the set of available node outputs is not decodable. We demonstrate the potential applications of this framework in various contexts, such as large-scale matrix multiplication and black-box optimization. We present the implementation of these methods on a serverless cloud computing system and provide numerical results to demonstrate their scalability in practice, including ImageNet scale computations.

1 Introduction

The utilization of distributed computing has become a crucial aspect in various scientific and engineering applications that involve the manipulation of large-scale data and models. Despite its advantages, distributed computing poses several challenges in algorithm design,

[*]Part of this manuscript have been presented in part as conference proceedings in 2019 Allerton Conference on Communication, Control, and Computing (see [1], [2]).

[†]B. Bartan is with the Dept. of Electrical Engineering, Stanford University, CA (bbartan@stanford.edu).

[‡]M. Pilanci is with the Dept. of Electrical Engineering, Stanford University, CA (pilanci@stanford.edu).

including inter-node communication, handling of malfunctioning or slow nodes, and maintaining data privacy.

In distributed computing, the presence of straggling nodes can significantly impact the overall computation time of an algorithm. To overcome the problem of stragglers, the idea of adding redundancy to computations using error-correcting codes has been explored in the literature by many recent works [3], [4], [5]. Error-correcting 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 robust distributed computing framework.

In this study, we propose a novel method that incorporates the principles of anytime computing [6] with coded computation and randomized sketching. The anytime computing approach allows for the generation of approximate solutions that improve in accuracy over time. Our method utilizes this concept by enabling exact recovery when the available outputs are decodable. In situations where the available outputs are not decodable, our method can provide accurate unbiased estimates for the desired computation at any point in time. These anytime estimates can be applied in machine learning and optimization as they can provide sufficient inexact gradients for these applications.

Our approach incorporates the use of polar codes for computation [1] and randomized sketching to achieve provable improvements in performance. Polar coding, a method for error-correcting code construction, has been shown to achieve the capacity of symmetric binary-input discrete memoryless channels [7]. As a result, we propose a flexible distributed computing method that is highly robust against stragglers and able to provide accurate approximations when exact recovery fails due to the high number of stragglers. The formal statement of our main result could be found in Theorem 2.2.

Serverless computing is a novel cloud-based computational model that enables users to execute computations within the cloud environment without the need for provisioning or managing servers. In order to ensure straggler resilience within this model, it is essential for the scheme to be scalable, as the number of nodes may vary greatly. As the number of worker nodes increases to the scale of hundreds or thousands, two considerations become particularly salient. The first one is that encoding and decoding of the code 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 as we will show in the sequel. They have low encoding and decoding complexity and both encoding and decoding require only a small number of subtraction and addition operations without any multiplication operations. In addition, they are known to achieve channel capacity in communication ([7]). The importance of this fact for coded computation is that we assume the outputs of worker nodes are analogous to binary erasure channels and thus the number of worker outputs needed for decoding is asymptotically optimal.

The distinction between serverless and server-based computing is crucial in understanding the utility of polar coding based approaches. Serverless computing requires a larger number of compute nodes to perform equivalent computation compared to server-based computing. This design consideration highlights the need for efficient encoding and decoding algorithms to address the increased computational requirements. For instance, for a distributed server-based system with $N = 8$ nodes, using maximum distance separable (MDS) codes with decoding complexity as high as $O(N^3)$ can be computationally feasible. However, this becomes a limitation when transitioning to a serverless system, as the number of functions N required to achieve the same level of computation may increase to several thousand. The reason for this is the limited resources such as low RAM and short lifetime that each function is restricted to have in serverless computing. Hence, it is imperative to employ codes with efficient decoding algorithms, such as polar codes. Despite lacking the MDS properties, polar codes exhibit a diminishing performance gap in terms of recovery threshold as the block-length of the code increases.

1.1 Related Work

Coded matrix multiplication has been introduced in [3] for speeding up distributed matrix-vector multiplication in server-based computing platforms. In [3], it was shown that it is possible to speed up distributed matrix multiplication by using MDS codes. 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 [4] attacks this problem by introducing a coded computation scheme based on d-dimensional product codes. [5] 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. [8] and [9] are other works investigating coded matrix multiplication and provide analysis on the optimal number of worker outputs required. The polynomial code approach has been extended in [10], where a secure, private, batch distributed matrix multiplication scheme has been proposed.

The matrix multiplication computation is partitioned into sequentially computed layers of varying precision in [11]. Coded computing is then applied to each precision layer to make the computation of each layer straggler-resilient. [12] considers the variability in the computation speed across the worker nodes. The state of each worker node is modeled using Markov chains and a dynamic computation strategy is developed. This work also has a coded computing aspect which is based on an MDS code. Our polar code approach could be applied to [12] to lower the complexity required for decoding in order to enable it for large-scale distributed computing. A common ingredient in both [11] and [12] is the use of finite field data and high complexity decoding (typically cubic complexity). Fast encoding and decoding procedures along with the approximation property of our method enable massive scale computing. [13] proposes an approximate coded computing algorithm with low complexity that act on real-valued data. Authors of [13] provide theoretical guarantees for the approximation quality. Our method not only exhibits similar desirable properties but also allows for fast exact decoding.

There are works in the literature on coded computation for gradient coding and different types of large-scale linear algebra operations such as [14], [15], [16]. Straggler mitigation in distributed computing with heterogeneous compute nodes is studied in [17]. In addition to the coding theoretic approaches, [18] presents an approximate straggler-resilient matrix multiplication scheme where sketching and straggler-resilient distributed matrix multiplication are combined.

Polar codes for coded computation is further studied in the work [19]. This work analyzes the convergence properties of polarization of computation times.

Using Luby Transform (LT) codes, a type of rateless fountain codes, in coded computation has been proposed in [20] and [21]. The proposed scheme in [21] divides the overall task into smaller tasks of multiplication of rows of A with x for better load-balancing. The work [20] proposes the use of inactivation decoding and the work [21] uses peeling decoder. Peeling decoder has a computational complexity of $O(N \log N)$, however its performance is not satisfactory if the number of input symbols is not very large. Inactivation decoder performs better than the peeling decoder in terms of error correction, however, it is not as fast as the peeling decoder.

Several alternative methods were proposed for designing anytime algorithms in distributed computing, capable of generating estimates and exact solutions with sufficient time ([22, 23, 24]). However, it is not guaranteed that an algorithm that provides exact recovery is computationally efficient. Our use of a randomized version of polar codes, which have efficient decoding algorithms as well as strong concentration properties, makes this possible. Our approach is novel in that it enables efficient exact recovery while also offering anytime inexact solutions with theoretical guarantees as outlined in Theorem 2.2.

Among the recent work on serverless computing for machine learning training is [25], where serverless machine learning is discussed in detail, and challenges and possible solutions on serverless machine training are provided. Resource allocation and pricing aspects of serverless computing are investigated in [26]. Authors in [27] 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 [28] shows via experiments that their prototype on AWS Lambda can reduce model training time greatly. A distributed convex optimization mechanism based on randomized second order optimization is proposed and studied in [29] for serverless computing.

We also investigate the application of black-box optimization methods for reinforcement learning. The work of [30] considers the evolution strategies method in reinforcement learning and show that distributed training with evolution strategies can be very fast because of its scalability. The work of [31] shows that using orthogonal exploration directions leads to lower errors and present the structured evolution strategies method which is based on a special way of generating random orthogonal exploration directions, as we discuss later in detail.

1.2 Overview of Our Contributions

- We introduce a novel approach for distributed computation of linear operations that is resistant to slow or "straggler" nodes while also providing approximate solutions when exact decoding is not possible. The method unifies polar codes and randomized Hadamard sketches to achieve this goal. This allows for robustness and flexibility in computation and highly efficient fast decoding, making it a versatile solution for large scale linear operation computations.

- We present methods for coded computation and black-box optimization using polar codes. We develop efficient algorithms for encoding and decoding over real numbers.

- We have extended the previous results on polarization of computation times into kernels of arbitrary size. Arbitrary size kernels could be useful when the number of nodes is not a power of 2.

- We present numerical results on large-scale data including ImageNet ([32]) that show the scalability of the proposed methods. We have implemented and tested the methods for the serverless computing service AWS Lambda.

2 Coded Computation using Polar Codes

2.1 Problem Setup

In this section, we will use the distributed computation of the matrix-vector multiplication operation Ax as a motivating example for the discussion. Suppose that $A \in \mathbb{R}^{n \times d}$ is a large data matrix partitioned to s sub-matrices A_i of size $\frac{n}{s}$-by-d over its rows:

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_s \end{bmatrix}, \text{ where } A_i \in \mathbb{R}^{\frac{n}{s} \times d}, \quad i = 1, \ldots, s. \tag{1}$$

To keep the presentation simple, we will assume that $x \in \mathbb{R}^d$ is a vector of manageable size and is not partitioned or encoded. For a detailed discussion of the general setting where x is a matrix and also encoded, the reader is referred to Appendix D. Our goal is to compute exactly or approximately the product Ax using N worker nodes that run in parallel. In our model, worker nodes are allowed to communicate only with the central node. We will assume the encoded data blocks are denoted as $\tilde{A}_i$:

$$\tilde{A} = \begin{bmatrix} \tilde{A}_1 \\ \vdots \\ \tilde{A}_N \end{bmatrix}, \text{ where } \tilde{A}_i \in \mathbb{R}^{\frac{n}{s} \times d}, \quad i = 1, \ldots, N. \tag{2}$$

The output of worker node i is then $\tilde{A}_i x$. Figure 1 presents a visual representation of the computing model. When worker node i finishes its assigned computation, we say that its output is available. The set of nodes whose outputs are available will be denoted as $\mathcal{S} \subseteq \{1, \ldots, N\}$. We will assume that the worker node outputs can be either "unavailable"

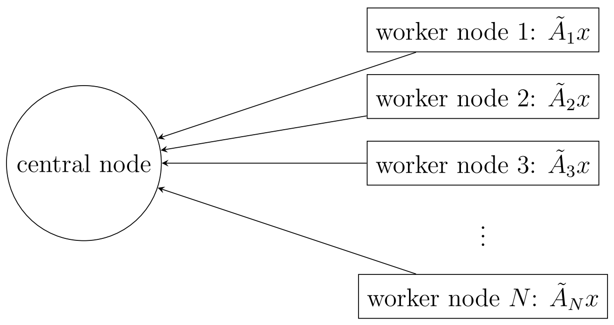

central node
worker node 1: $\tilde{A}_1x$
worker node 2: $\tilde{A}_2x$
worker node 3: $\tilde{A}_3x$
worker node N: $\tilde{A}_Nx$

Figure 1: Distributed computing model.

or "available and correct". In other words, if an output is available, we will assume it is error-free. Consequently, the worker nodes can be modeled as real-valued erasure channels.

2.2 Main Result

Our primary contribution is a novel technique for introducing redundancy in computation that effectively eliminates the straggler effect in exact recovery and at the same time, provides approximation guarantees for the anytime estimates. Our scheme enables finding an unbiased estimator for the matrix-vector product Ax that provides guaranteed approximation results when the set of available outputs is not sufficient for exact decoding of the result. An essential aspect of our analysis is the synthesis of two distinct areas of research: Polar codes and the concentration of measure for randomized Hadamard sketches.

The proposed method is summarized in Algorithm 1. Our encoder acts on the blocks A_i of the data matrix A and returns the encoded data blocks $\tilde{A}_i$. Details of the encoding procedure are given in Section 2.3. The worker nodes are assigned tasks such that worker node i computes the product $\tilde{A}_ix$. Then, the central node monitors the available node outputs and updates the set $\mathcal{S}$ and approximation of the computation accordingly. Once $\mathcal{S}$ becomes decodable (see Definition 2.1), the decoder recovers the desired result Ax exactly. We will provide the details of the encoding and decoding algorithms in the sequel.

Definition 2.1 (Decodable set)**.** *Let $\mathcal{S}$ denote the set of nodes that complete their assigned computation. We say $\mathcal{S}$ is decodable whenever it is possible to exactly recover the desired* *result from the outputs of nodes in $\mathcal{S}$ using our sequential decoding procedure (Algorithm 2).*

Algorithm 1: Randomized polar coding based anytime distributed computing

```
**Input:** Data $A, x$
$\tilde{A} = \text{encoder}(A)$ (see Section 2.3.1)
Assign $\tilde{A}_i x$ to worker node $i$, $i = 1, \ldots, N$
Initialize $\mathcal{S} = \{\}$
**while** $\mathcal{S}$ *not decodable* **do**
    update $\mathcal{S}$
    form the approximate anytime estimator $\mathcal{T}_x(\mathcal{S}) := \text{vec}\left(\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} z_i(\tilde{A}_i x)^T\right)$
$Ax = \text{decoder}(\mathcal{S})$ (see Algorithm 2)
return $Ax$ (exact computation)
```

We defer the analysis of exact decodability via sequential decoding to Section 2.6. When $\mathcal{S}$ is not decodable, we will show that it is possible to construct approximate solutions using the estimator $\mathcal{T}_x(\mathcal{S})$ defined as

$$\mathcal{T}_x(\mathcal{S}) := \text{vec}\left(\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} z_i(\tilde{A}_i x)^T\right) \in \mathbb{R}^n \tag{3}$$

where $z_1, ..., z_{|\mathcal{S}|} \in \mathbb{R}^s$ are column vectors corresponding to the $|\mathcal{S}|$ rows of the randomized polar code matrix $Z = [z_1, ..., z_N]^T$ as defined in Section 2.3.1 that correspond to the nodes that complete their assigned computation. In the above formula, the vec$(\cdot)$ operation vectorizes the $s \times \frac{n}{s}$ matrix $\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} z_i(\tilde{A}_i x)^T$ columnwise into a size-n vector to provide an anytime estimator for $Ax \in \mathbb{R}^n$.

We now present our main result on the quality of the anytime estimator.

Theorem 2.2 (Main result). *Suppose that the worker run times are independently distributed. Algorithm 1 returns the matrix-vector product Ax exactly for decodable*[1] *$\mathcal{S}$ in $\mathcal{O}(N \log N)$ decoding time. In the case when $\mathcal{S}$ is not decodable, the approximation* (3)

[1]The distribution of the time at which exact computation is possible can be found in closed form as we show in Section 2.6

is unbiased and for any fixed collection of vectors x, x', c we have

$$(1-\epsilon)\|A(x-x')\|_2^2 \leq \langle \mathcal{T}_x(\mathcal{S}) - \mathcal{T}_{x'}(\mathcal{S}), A(x-x')\rangle \leq (1+\epsilon)\|A(x-x')\|_2^2, \tag{4}$$

and

$$-\epsilon(\|c\|_2^2 + \|Ax\|_2^2) \leq \langle c, \mathcal{T}_x(\mathcal{S}) - Ax\rangle \leq \epsilon(\|c\|_2^2 + \|Ax\|_2^2), \tag{5}$$

with probability at least $1 - s\exp(-C_2|\mathcal{S}|)$ when at least $C_1 \log(N)^4/\epsilon^2$ workers finish their computation[2]*. Here, $\mathcal{T}_x(\mathcal{S})$ and $\mathcal{T}_{x'}(\mathcal{S})$ are approximations of Ax and Ax' respectively. In addition, the estimator $\mathcal{T}_x(\mathcal{S})$ can be computed in $\mathcal{O}(N \log N)$ time.*

The proof is presented in Sections 2.4.1 and 2.5. The bound (5) shows that the inner-product $c^T Ax$ can be replaced with $c^T\mathcal{T}_x(S)$ with small error for any fixed vector c. A corollary of the approximation in (4) is that $1-\epsilon \leq \langle \mathcal{T}_x(\mathcal{S}), Ax\rangle \leq 1+\epsilon$ for any x satisfying the normalization $\|Ax\|_2 = 1$. The main significance of Theorem 2.2 is that it shows there exists a strategy of encoding data that allows for unbiased estimates at anytime and eventual exact recovery. Moreover, the complexity of encoding and decoding with exact recovery, and also approximate recovery is only $\mathcal{O}(N \log N)$, which is significantly faster than standard computational codes [3].

2.3 Encoding

Here we describe the randomized polar coding construction underlying our computational scheme. We will use $U_1, \ldots, U_N$ to denote the input data blocks to the encoder. Some of the inputs will be set to zero and others will be set to the data blocks A_i. This is to control the redundancy of the code, analogous to the frozen and data bits in traditional polar codes.

Consider the construction given in Figure 2 for $N = 2$ data blocks. The rectangles M_1 and M_2 correspond to worker nodes 1 and 2. The input blocks U_1 and U_2 are first

[2]C_1, C_2 are constants independent of dimensions.

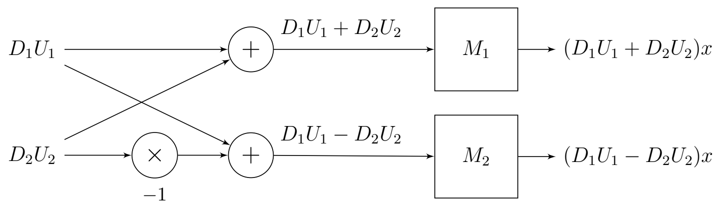


Figure 2: Randomized polar code construction for $N = 2$.

multiplied by the diagonal matrix $D = \mathrm{Diag}(D_1, D_2)$ where the diagonal entries D_i are sampled from a scalar Rademacher distribution with ± 1 i.i.d. entries. This is then input to the Hadamard kernel. The outputs are the encoded data blocks and equal to $D_1U_1 + D_2U_2$ and $D_1U_1 - D_2U_2$. The worker nodes are tasked with multiplying these encoded blocks by x. The variables D_i are generated only once during the construction of the code and kept fixed during the decoding.

Remark 2.3. *Note that the encoding shown in Figure 2 is for the Hadamard kernel* $\left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$, *whereas the polar code kernel is given by* $\left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$. *Encoding and exact decoding procedures work similarly for both kernels. However, approximate recovery is only possible via the Hadamard kernel due to the connection to randomized Hadamard sketches as shown in the sequel.*

Larger size constructions can be obtained by recursively applying the size 2 constructions. For instance, the construction for $N = 4$ is shown in Figure 3. The recursive construction enables fast encoding and decoding operations, making it suitable for large-scale computing.

We now discuss the procedure for determining which inputs U_i to freeze, i.e. set to zero, and which ones to send in data blocks. Let us denote the erasure probability of each worker node by ϵ, and assume that the erasures are independent. The calculation of the erasure probabilities for the transformed nodes is similar to traditional polar codes, which we show with the example of $N = 2$: Suppose that a sequential decoder in the first stage recovers D_1U_1 from the outputs of M_1 and M_2 in Figure 2, and then recovers D_2U_2 given D_1U_1 in the second stage. The probability that the first stage fails is given by

$$\mathbb{P}[\text{output } M_1 \text{ is erased } \textbf{or} \text{ output } M_2 \text{ is erased}] = F_+(\epsilon) := 1 - (1-\epsilon)^2,$$

since any erasure makes the recovery of D_1U_1 impossible. On the other hand, the probability that the second stage fails is given by

$$\mathbb{P}[\text{output } M_1 \text{ is erased } \textbf{and} \text{ output } M_2 \text{ is erased}] = F_-(\epsilon) := \epsilon^2,$$

since D_2U_2 can be recovered either from the output of M_1 or M_2 given the knowledge of D_1U_1. For larger construction sizes, the above calculation can be extended recursively. For $N = 4$, the erasure probabilities of the transformed nodes are given by

$$\Big\{F_+(F_+(\epsilon)), F_+(F_-(\epsilon)), F_-(F_+(\epsilon)), F_-(F_-(\epsilon))\Big\},$$

(see e.g., [1]). Based on the erasure probabilities of the transformed nodes, we select the best ones for data, and freeze the rest (i.e., set to zero matrices). These transformed nodes are analogous to virtual channels in polar coding for communication [7].

After computing the erasure probabilities for the transformed nodes, we choose the $N(1-\epsilon)$ nodes with the lowest erasure probabilities as data nodes. The remaining $N\epsilon$ nodes are frozen. For example, for $N = 4$ and $\epsilon = 0.5$, the erasure probabilities of the transformed nodes are calculated to be $\{0.938, 0.563, 0.438, 0.063\}$. It follows that we freeze the first two inputs, and the last two inputs are set to data blocks. This means that in Figure 3, we set $U_1 = U_2 = 0^{n/2\times d}$ and $U_3 = A_1$, $U_4 = A_2$.

Note that unlike the XOR operation of polar codes in binary communication channels, in this work we consider real numbers and linear polarizing transformations over real numbers. As it will be shown in the sequel, the channel polarization phenomenon in finite fields carries over to the reals in an analogous manner. We note that the encoding procedure has computational complexity $O(N \log N)$. This can be seen by observing that there are $\log_2 N + 1$ vertical levels in the code construction and N nodes in every level.

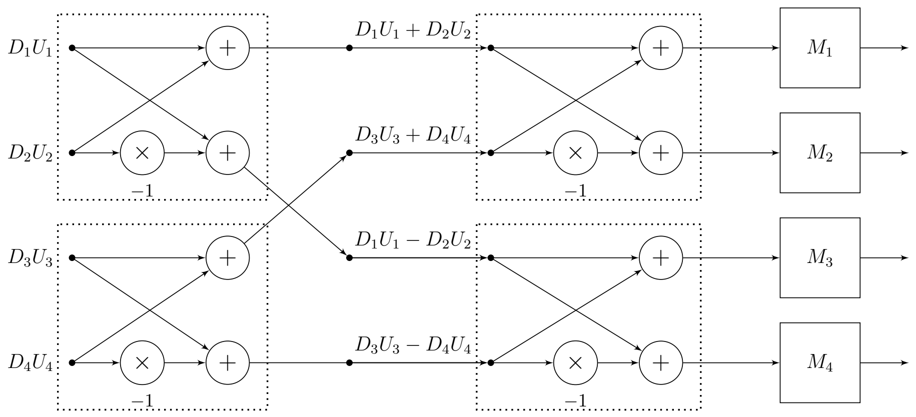


Figure 3: Randomized polar code construction for $N = 4$.

2.3.1 Randomized Polar Code Construction

We define the following randomized linear code

$$Z := HDR\,, \tag{6}$$

where H is the $N \times N$ Hadamard matrix, $D \in \mathbb{R}^{N\times N}$ is a diagonal matrix containing uniform ± 1 Rademacher random variables, and $R \in \mathbb{R}^{N\times s}$ is a 0–1 matrix whose certain rows are set to zeros. The matrix R pads zero entries to frozen data locations, which are determined according to polarized erasure probabilities. The matrix R can be constructed by taking an $s \times s$ dimensional identity matrix and padding all-zero rows at the frozen data indices.

For the case of $s = n$, the encoded data is defined as $\tilde{A} := ZA$. For the general case of matrix data $A \in \mathbb{R}^{n\times d}$, we partition A into blocks of size $\frac{n}{s} \times d$ as $A = [A_1; \ldots; A_s]$, where the semicolons indicate that blocks are stacked vertically. The encoded data is defined as

$$\begin{aligned} &\tilde{A} = \text{encode}(A) := [\tilde{A}_1; \ldots; \tilde{A}_N] \in \mathbb{R}^{Nn/s\times d}, \text{ where} \\ &[\text{vec}\,(\tilde{A}_1), \ldots, \text{vec}\,(\tilde{A}_N)]^T := Z[\text{vec}\,(A_1), \ldots, \text{vec}\,(A_s)]^T\,, \end{aligned} \tag{7}$$

where vec$(\cdot)$ is the columnwise vectorization operator. In the case of the encoded matrix-vector product task, the workers are assigned to compute $\tilde{A}x$. The encoding function can be equivalently rewritten using the Kronecker product as follows:

$$\tilde{A} = \text{encode}(A) := (Z \otimes I_{n/s})A\,, \tag{8}$$

where $I_{n/s}$ is the $(n/s) \times (n/s)$ dimensional identity matrix.

2.4 Approximate Computation

The Hadamard matrix is also used for dimension reduction in randomized approximate algorithms, including the well-known Subsampled Randomized Hadamard Transform (SRHT) [33, 34]. Until this work, the connection between polar codes and SRHT has not been understood. SRHT is constructed as $S_H := \frac{1}{\sqrt{m}} PHD \in \mathbb{R}^{m\times N}$ where $P \in \mathbb{R}^{m\times N}$ is a $0-1$ row-sampling matrix that picks m rows uniformly at random, $H \in \mathbb{R}^{N\times N}$ is the Hadamard matrix with orthonormal columns, and $D \in \mathbb{R}^{N\times N}$ is a diagonal matrix with diagonal entries sampled i.i.d. from the Rademacher distribution, $D_{ii} = -1$ or 1 with probability 1/2. For $m < N$, the sketched data matrix $S_H A = \frac{1}{\sqrt{m}} PHDA \in \mathbb{R}^{m\times d}$ can be used as an approximate low dimension version of the data matrix.

We can rewrite the expression for the estimator given in (3) equivalently as follows:

$$\begin{aligned} \mathcal{T}_x(\mathcal{S}) &= \text{vec}\left(\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}} z_i z_i^T [A_1x, ..., A_sx]\right) \\ &= \text{vec}\left(\frac{1}{|\mathcal{S}|} DHP^TPHD[A_1x, ..., A_sx]\right) \\ &= \begin{bmatrix} S_H^T S_H A_1 x \\ \dots \\ S_H^T S_H A_s x \end{bmatrix} \end{aligned} \tag{9}$$

where $P \in \mathbb{R}^{m\times N}$, $m = |\mathcal{S}|$ is a uniform row subsampling matrix that encodes the $|\mathcal{S}|$ workers that finished computation. Therefore, each column of $DHP^TPHD[A_1x, ..., A_sx]$ is identical to an SRHT sketch applied simultaneously to the s blocks of the desired matrix product

$A_1x, ..., A_sx$, where the sketch size equals to $|\mathcal{S}|$.

As the first step of showing that the estimator provides good approximations for the true result, we state Lemma 2.4, which shows that the approximate results that we obtain using the estimator are unbiased estimates.

Lemma 2.4 (Unbiasedness)**.** *Suppose that the worker job completion times are i.i.d. Then, the estimator $\mathcal{T}_x(\mathcal{S})$ provides unbiased estimates of the true result, i.e., $\mathbb{E}[\mathcal{T}_x(\mathcal{S})] = Ax$, where the randomness of the expectation is with respect to the diagonal Rademacher matrix D and the randomness in the job completion times of the workers.*

Proof. Consider the expectation of the estimator $\mathcal{T}_x(\mathcal{S})$ where the randomness is with respect to the randomness of D:

$$\begin{aligned}\mathbb{E}[\mathcal{T}_x(\mathcal{S})] &= \mathbb{E}\,\mathrm{vec}\left(\frac{1}{|\mathcal{S}|}DHP^TPHD[A_1x, ..., A_sx]\right)\\ &= \mathrm{vec}\left(\frac{1}{|\mathcal{S}|}\sum_{i\in\mathcal{S}}\mathbb{E}_D\left[\frac{1}{|\mathcal{S}|}DH\,\mathbb{E}_P[P^TP]HD\right][A_1x, ..., A_sx]\right)\\ &= \mathrm{vec}\left([A_1x, ..., A_sx]\right) = Ax\,. \end{aligned} \tag{10}$$

Note that the third equality follows from the fact $\mathbb{E}_P[P^TP] = |\mathcal{S}|I$ due to the i.i.d. distribution of worker job completion times, since the matrix P is a row-sampling matrix whose each row is sampled i.i.d. with replacement. Finally, $H^2 = I$ and $\mathbb{E}\,D^2 = I$. □

2.4.1 Analysis of the anytime estimator

Now we give a proof of the anytime estimation guarantees presented in Theorem 2.2. By the linearity of the map $\mathcal{T}_x(S)$ with respect to x, we have $\mathcal{T}_x(S) - \mathcal{T}_{x'}(S) = \mathcal{T}_{x-x'}(S)$. Therefore, we need to prove that

$$(1-\epsilon)\|A(x-x')\|_2^2 \le \langle \mathcal{T}_{x-x'}(S), A(x-x')\rangle \le (1+\epsilon)\|A(x-x')\|_2^2.$$

Without loss of generality, we may assume $x' = 0$ and it suffices to show

$$(1-\epsilon)\|Ax\|_2^2 \leq \langle \mathcal{T}_x(S), Ax \rangle \leq (1+\epsilon)\|Ax\|_2^2,$$

for a fixed vector x.

Next, we present the following result on the Johnson-Lindenstrauss (JL) property of the map HD after random erasures, which is a consequence of the analysis of SRHT from the sketching literature [35]:

Theorem 2.5. *Suppose that $P \in \mathbb{R}^{m \times N}$ is an i.i.d. row sampling matrix, where $m \geq O(\log(N)^4)/\epsilon^2$. Then, for any fixed vector u, it holds with probability at least $1 - \exp(-mc_1)$ that*

$$(1-\epsilon)\|u\|_2^2 \leq \|PHDu\|_2^2 \leq (1+\epsilon)\|u\|_2^2\,,$$

where C_1 is a fixed positive constant.

Here, the i.i.d. row-sampling matrix P is achieved by the i.i.d. job completion times of the workers. This result implies that the encoding a vector u as HDu followed by random erasures preserves ℓ_2 norms even though exact recovery may not be possible. Note that for any fixed vector u, we have

$$\begin{aligned} \|PHDu\|_2^2 - \|u\|_2^2 &= u^T(PHD)^T PHDu - \|u\|_2^2 \\ &= u^T S_H^T S_H u - \|u\|_2^2\,. \end{aligned} \tag{11}$$

It follows from Theorem 2.5 that $|u^T S_H^T S_H u - \|u\|_2^2| \leq \epsilon \|u\|_2^2$ with high probability. Next, we apply this bound to bound bilinear terms $u^T S_H^T S_H u'$ for any fixed u and u'. We combine the last bound with the identity

$$|u^T Q u'| = \frac{1}{2}|(u+u')^T Q (u+u') - u^T Q u - u'^T Q u'|,$$

which is valid for any symmetric matrix Q by applying (11) three times to obtain

$$|u^T(S_H^T S_H - I)u'| \leq \frac{3}{2}\epsilon(\|u\|_2^2 + \|u'\|_2^2). \tag{12}$$

Next, we apply the high-probability bound $|u^T S_H^T S_H u - \|u\|_2^2| \leq \epsilon\|u\|_2^2$ to each data block by letting $u = A_i x$ for $i = 1, ..., s$. We obtain

$$\mathbb{P}\Big[\exists i \in [\mathcal{S}] \text{ s.t. } |(A_i x)^T S_H^T S_H (A_i x) - \|A_i x\|_2^2| > \epsilon\|A_i x\|_2^2\Big] \leq s\exp(-mC_1),$$

where the inequality follows from the union bound and Theorem 2.5. Here, C_1 is a fixed constant. Finally, adding these inequalities we obtain with high probability the following

$$(1-\epsilon)\sum_{i=1}^{s}\|A_i x\|_2^2 \leq \sum_{i=1}^{s}(A_i x)^T S_H^T S_H (A_i x) \leq (1+\epsilon)\sum_{i=1}^{s}\|A_i x\|_2^2\,.$$

We have $\|Ax\|_2^2 = \sum_{i=1}^{s}\|A_i x\|_2^2 = 1$ and therefore

$$(1-\epsilon)\|Ax\|_2^2 \leq \langle \mathcal{T}_x(S), Ax\rangle \leq (1+\epsilon)\|Ax\|_2^2\,,$$

with probability at least $s\exp(-mC_1)$, which proves the first inequality of Theorem 2.2. Therefore, the inner product between the estimate $\mathcal{T}_x(\mathcal{S})$ and the vector Ax is close to $\|Ax\|_2^2$ with high probability.

Next, for fixed arbitrary vectors $\{c_i\}_{i=1}^{s}$ and x we apply (12) to each data block as follows

$$\mathbb{P}\Big[\exists i \in [\mathcal{S}] \text{ s.t. } |c_i^T S_H^T S_H A_i x - c_i^T A_i x| > \frac{3}{2}\epsilon(\|A_i x\|_2^2 + \|c_i\|_2^2)\Big] \leq s\exp(-mC_1).$$

Note that we can scale ϵ and absorb the constant factors into the constant C_1. Adding these inequalities, we obtain with high probability the following

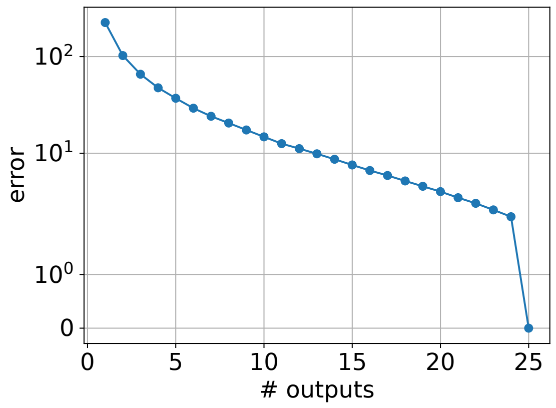


Figure 4: Error as a function of the number of outputs used in computing the estimate. The data matrix $A \in \mathbb{R}^{9600\times 1000}$ is synthetically generated by sampling from the standard normal distribution. The vector $x \in \mathbb{R}^{1000}$ is also sampled from the standard normal distribution and then scaled by 10^{-3}. The vertical axis shows the squared ℓ_2 norm between Ax and the estimate. The number of worker nodes is $N = 32$. 24 inputs are used for data and the remaining 8 are frozen. For this simulation, the set of available nodes becomes decodable after 25 nodes return their output.

$$|c^T(\mathcal{T}_x(S) - Ax)| \leq \epsilon(\|Ax\|_2^2 + \|c\|_2^2) \tag{13}$$

with probability at least $\exp(-mC')$ where $c = [c_1^T, ..., c_s^T]^T$ and C' is a fixed constant. This proves the second inequality of the theorem. Finally, note that $\mathcal{T}_x(S)$ can be computed in $\mathcal{O}(N \log N)$ time using the Fast Hadamard Transform [7].

The empirical performance is illustrated for $N = 32$ in Figure 4 for a synthetically generated dataset. The error decreases as the number of outputs increases until a decodable set of outputs is detected.

It is worth noting that our approximate recovery result can be viewed as a unification of sketching and coding interpretations of the Hadamard matrix. Specifically, our construction combines the diagonal random Rademacher matrix D from SRHT with the polarization phenomenon from polar codes. As a consequence, we obtain JL embeddings with erasure recovery properties at $O(N \log(N))$ encoding and decoding time. Moreover, since our construction is based on polar codes and sequential decoding, they inherit their capacity achieving properties under random erasures.

2.5 Exact Recovery via Sequential Decoding

In the previous subsection, we have focused on finding an approximate result when $\mathcal{S}$ is not decodable. Now, we discuss the exact recovery of Ax via sequential decoding.

Algorithm 2: Decoding algorithm

```
Input: The set $\mathcal{S}$ and indices of the frozen inputs
Initialize $I_{i,j}$ = False for all $i \in [1, N]$, $j \in [1, \log N + 1]$
for $i \in \mathcal{S}$ do
    $I_{i,\log N+1}$ = True
Initialize an empty list $y$
for $i \leftarrow 1$ to $N$ do
    $\_, z_{i,1}$ = decodeRecursive($i$, 1)
    if node $i$ is not frozen then
        $y = [y; z_{i,1}]$
    if $i \bmod 2 = 0$ then                          ▷ forward propagation
        for $j \leftarrow 1$ to $\log N + 1$ do
            for $l \leftarrow 1$ to $i$ do
                compute $z_{l,j}$
return $y = Ax$
```

The decoding algorithm for exact recovery is given in Algorithm 2. The decoder is a sequential algorithm that performs recovery one at a time and in an order. The notation $I_{i,j} \in \{True, False\}$ indicates whether we know the value at node i in level j in the code construction circuit. *Level* indicates the horizontal position while *node* is the vertical position. $z_{i,j}$ is a data structure that holds the value for node i in level j.

The decoding algorithm has a subroutine called *decodeRecursive* given in Algorithm 3. The main idea behind the decoding algorithm is that it works recursively from right to left (i.e. from output to input) in the code construction. It performs decoding for 2×2 blocks independently and combines the result. The 2×2 blocks are shown as dashed rectangles in Figure 3. For instance, the top left dashed rectangle has inputs $z_{1,1}, z_{2,1}$ and outputs $z_{1,2}, z_{2,2}$. The mapping between the inputs and outputs is given by the Hadamard kernel: $z_{1,2} = z_{1,1} + z_{2,1}$ and $z_{2,2} = z_{1,1} - z_{2,1}$. Hence, when we wish to recover $z_{1,1}$, we sum the outputs and divide by 2. Similarly, to recover $z_{2,1}$, we subtract the second output from the first one and divide by 2.

In the algorithm, we use the term *pair* to refer to inputs or outputs for a single 2×2 block. In the above example, $z_{1,1}$ and $z_{2,1}$ are a pair and $z_{2,1}$ and $z_{2,2}$ are another pair. The notation $I_{\text{pair}(i),j}$ is used to refer to the other node in the pair that node i is in at the j'th level. For the above example, $I_{\text{pair}(1),1}$ is the same as $I_{2,1}$ since node 2 is in a pair with node 1 in the first level. Furthermore, we call the first node in the pair *upper node*. For instance, $z_{1,1}$ and $z_{1,2}$ are upper nodes in their respective pairs. The goal of the decoder algorithm is to compute the values for all the nodes in the first level $z_{i,1}$, $i = 1, \ldots, N$.

The step of multiplication of inputs U_i by the diagonal matrix D is necessary for the approximate recovery. In the case of exact recovery, the outputs of the decoding algorithm will be the terms $D_i U_i x$. The desired outputs $U_i x$ can be obtained via dividing by D_i's.

Algorithm 3: decodeRecursive(i, j)

```
Input: Node $i \in [1, N]$, level $j \in [1, \log N + 1]$
if $j = \log N + 1$ then
    return ($I_{i,j}$, $z_{i,j}$) if $I_{i,j}$ = True, else ($I_{i,j}$, None)        ▷ base case 1
if $I_{i,j}$ = True then
    return (True, $z_{i,j}$)                                                          ▷ base case 2
$I_{i,j+1}$ = decodeRecursive($i, j+1$)
$I_{\text{pair}(i),j+1}$ = decodeRecursive(pair($i$), $j+1$)
if $i$ is upper node then
    if $I_{i,j+1}$ AND $I_{pair(i),j+1}$ = True then
        compute $z_{i,j}$
        return (True, $z_{i,j}$)
else
    if $I_{i,j+1}$ OR $I_{pair(i),j+1}$ = True then
        compute $z_{i,j}$
        return (True, $z_{i,j}$)
return (False, None)
```

2.6 Polarization of Computation Times

In this section, we analyze the time at which the coded computations are decodable. We show that a real valued version of the recursive construction in polar codes enables polarization of the probability density functions towards better or worse computing times, analogous to perfectly noiseless or noisy channels.

2.6.1 Characterization of Polarizing Kernels

As opposed to classical Polar Codes that operate in finite fields, our constructions have more freedom in their design space over the real numbers. In particular, the polarization kernels can be chosen arbitrarily as long as polarization takes place. In this section, we provide a characterization of matrices that enable polarization and exact recovery. We first give the definition of a *polarizing kernel*, and then state Lemma 2.7 that characterizes the conditions for a polarizing kernel.

Definition 2.6 (Polarizing 2×2 kernel over reals). *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. Note that this definition exclusively considers the earliest time at which $f(u_i)$ can be computed, without accounting for the decoding time involved.*

Lemma 2.7. *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 non-zero, 2) K is invertible.*

Theorem 2.8 builds on Lemma 2.7 to identify the polarizing kernels that require the least amount of computations for encoding.

Theorem 2.8. *Of all possible* 2×2 *polarizing kernels, the kernels* $F_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ *and* $F_2' = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ *result in the fewest number of computations for encoding real-valued data.*

The proofs of Lemma 2.7 and Theorem 2.8 are in the Appendix. Note that the results so far apply to only size 2 kernels. Next, we extend the results to kernels of bigger size.

Definition 2.9 (Polarizing $p \times p$ kernel over reals)**.** *This definition extends Definition 2.6 for polarizing kernels to arbitrary size kernels. Let* π_i *denote the index of the node* i *when the run times of nodes are sorted in decreasing order,* $T_{\pi_1} \geq T_{\pi_2} \geq \cdots \geq T_{\pi_p}$*. If the time required to compute* $f(u_i)$ *is equal to* T_{π_i} *for all* $i = 1, \ldots, p$*, then we say it is a polarizing kernel.*

(i) Kernel Size 3: Consider the 3×3 kernel $K = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$. Then, K is a polarizing kernel if and only if

1. K is invertible over the reals
2. All of the matrices $\begin{bmatrix} b & c \\ e & f \end{bmatrix}$, $\begin{bmatrix} b & c \\ h & i \end{bmatrix}$, $\begin{bmatrix} e & f \\ h & i \end{bmatrix}$ are invertible over the reals
3. c, f, i are all non-zero.

We note that the following kernel does not require any multiplications and only requires additions and subtractions and also is a polarizing kernel: $K = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$.

(ii) Arbitrary Kernel Size: Let K be a $p \times p$ kernel. It is a polarizing kernel if and only if it satisfies the following conditions:

1. K is invertible
2. After removing the first column of K, every $p-1$ rows of the remaining matrix is a matrix invertible over the reals
3. After removing the first and second columns of K, every $p-2$ rows of the remaining matrix is a matrix invertible over the reals

$\vdots$

$p-1$. After removing the first $(p-1)$ columns of K, every scalar in the remaining matrix is non-zero (i.e. 1×1 invertible matrix).

Note that for $p = 4$, the following upper triangular matrix is a polarizing kernel:

$$K = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

2.6.2 Recursive Polarization

As in polar codes, we now consider the recursive application of any 2×2 polarizing kernel via the Kronecker power construction $K_{2N} = K \otimes K_N$ and $K_2 = K$. For instance, one can take $K = F_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ to obtain the real valued version of polar codes. This operation is depicted for F_4 in Figure 3. Note that the input is permuted to bit reversed order in Figure 3. We refer the reader to [7] for a detailed description of the bit reversal process. In the case of the Hadamard kernel H_2, the recursive construction coincides with the Hadamard transformation. We now analyze the run-times in the recursive construction by illustrating the 4×4 construction.

Lemma 2.10. *Suppose that the kernel K is polarizing as given in Definition 2.6 and Lemma 2.7, and let $T_1, \ldots, T_4$ be random i.i.d. run-time random variables. Then the sequential decoding procedure described in Algorithm 2 computes $f(u_1), f(u_2), f(u_3), f(u_4)$ respectively in time*

$$T^{(1)} = \max\big(\max(T_1, T_2), \max(T_3, T_4)\big)$$
$$T^{(2)} = \min\big(\max(T_1, T_2), \max(T_3, T_4)\big)$$
$$T^{(3)} = \max\big(\min(T_1, T_2), \min(T_3, T_4)\big)$$
$$T^{(4)} = \min\big(\min(T_1, T_2), \min(T_3, T_4)\big).$$

Proof. Note that the recursive construction (e.g. in Figure 3) combines two independent random run-times and transforms to $\max(T_1, T_2)$ and $\min(T_1, T_2)$. Recursively applying this transformation proves the statement. $\square$

Another way to show the run-time transformations for $N = 4$ would be to consider that to recover the first input, we need to know both outputs of the top left dashed rectangle in Figure 3. That requires knowing all 4 outputs; hence $T^{(1)}$ is equal to maximum of all T_i's. Next, given the first input, to recover the second one, we need only one of the outputs of the top left dashed rectangle. Therefore, we obtain that $T^{(2)} = \min\big(\max(T_1, T_2), \max(T_3, T_4)\big)$. The idea is similar for the recovery of the third and fourth inputs.

The result of Lemma 2.10 naturally extends to constructions of arbitrary sizes where the corresponding run times are alternating min and max expressions. Consequently, one can freeze certain variables to obtain a faster overall run-time. For example, freezing the first input by setting it to a fixed value (e.g., zero), the decoder leverages this knowledge to eliminate the run-time $T^{(1)} = \max\big(\max(T_1, T_2), \max(T_3, T_4)\big)$.

Example 1. Let us illustrate Lemma 2.10 for the case of uniformly random run time distributions with closed-form formulas. Suppose that T_1, T_2, T_3, T_4 are i.i.d. and uniform in the interval $[0, 1] \subseteq \mathbb{R}$. Then a straightforward calculation shows that the probability density functions of the run times $T^{(1)}, T^{(2)}, T^{(3)}, T^{(4)}$ are given by

$$\begin{aligned}
p_{T^{(1)}}(t) &= 4t^3 \\
p_{T^{(2)}}(t) &= 4t(1-t)(1+t) \\
p_{T^{(3)}}(t) &= 4t(1-t)(2-t) \\
p_{T^{(4)}}(t) &= 4(1-t)^3
\end{aligned}$$

for $t \in [0, 1]$. It can be seen that the probability density functions are degree $N - 1$ polynomials for the general size N construction with the uniform distribution.

Polarization for Kernel Size 3: Let us denote the inputs by x_1, x_2, x_3 and the corresponding channel outputs by y_1, y_2, y_3. Next, note that x_1 can be recovered when all three outputs y_1, y_2, y_3 are known. Hence the run time for x_1 is equal to $\max(T_1, T_2, T_3)$. Given x_1, decoding x_2 requires at least two of y_1, y_2, y_3. The run-time for x_2 is equal to $\text{median}(T_1, T_2, T_3)$. Given x_1, x_2, the decoding of x_3 will take $\min(T_1, T_2, T_3)$.

Note that the run times for a construction of size $N = 9$ can be obtained using the same method that we previously described for computing the run times for $N = 4$ with kernel size 2 in Lemma 2.10. In particular, the run times for $N = 9$ will be as follows: Decoding x_1 will take time $\max(\max(T_1, T_2, T_3), \max(T_4, T_5, T_6), \max(T_7, T_8, T_9)) = \max(T_1, \ldots, T_9)$. Decoding x_2 will take time $\text{median}(\max(T_1, T_2, T_3), \max(T_4, T_5, T_6), \max(T_7, T_8, T_9))$. Decoding the last input x_9 will take time $\min(T_1, \ldots, T_9)$.

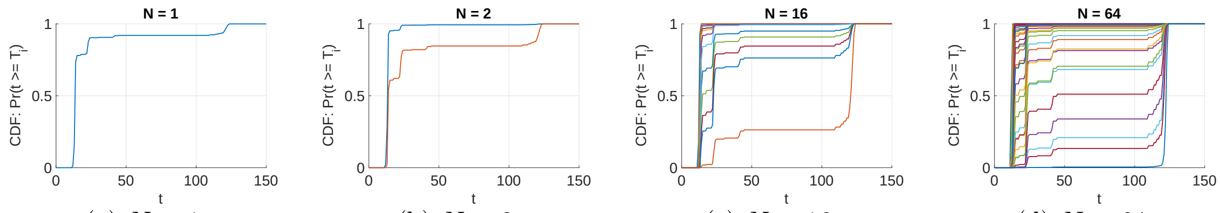
Polarization for Arbitrary Kernel Sizes: Let $p \geq 2$ denote the kernel size. Let π_i denote the index of the node i when the run times of nodes are sorted in decreasing order, $T_{\pi_1} \geq T_{\pi_2} \geq \cdots \geq T_{\pi_p}$. Then, for kernels of arbitrary size p, the run-time required for decoding x_i is equal to T_{π_i}. It is easy to observe that polarization for kernel sizes 2 and 3 are a special case of this result.

3 Numerical Results

In this section, we present numerical results to verify our theoretical claims and test the performance of the proposed methods in various tasks.

3.1 Polarization of Computation Times

Figure 5 is a visualization of the polarization of computation times. Plot (a) shows the empirical cumulative distribution function (CDF) of computation times for serverless functions in AWS Lambda. This empirical CDF has been obtained by running the same Python script in 500 serverless functions in parallel in AWS Lambda. Furthermore, observe that plot (a) shows that there are worker nodes that finish their computations much later in roughly



(a) $N = 1$ (b) $N = 2$ (c) $N = 16$ (d) $N = 64$

Figure 5: Empirical CDFs for various construction sizes show the polarization of run times.

$t = 120$ seconds as opposed to many worker nodes that finish before $t = 20$ seconds.

Plot (b) of Figure 5 shows the CDF for the transformed computation times for 2 workers. This has been simulated by assuming there are 2 worker nodes and they have i.i.d. computation times with CDF shown in plot (a). Similarly, plot (c) and (d) show the CDFs for 16 and 64 worker nodes. This process shows that we transform the computation times into better and worse computation times, i.e. polarization of the computation times. Freezing the inputs with worse computation times leads to a straggler-resilient computation mechanism since this is the same as picking only the transformed nodes with better computation times to perform the actual computation.

3.2 Gradient Descent for the Least Squares Problem

Polar coded distributed computation method can be used in any algorithm that requires matrix multiplication. Consider the gradient descent algorithm being applied to solve a linear least squares problem $\text{minimize}_x \|Ax - y\|_2^2$, where A is a large-scale data matrix. The variable x is small enough to fit in the memory of a worker node. The update rule for gradient descent is as follows:

$$x_{t+1} = x_t - \mu(A^T A x_t - A^T y), \tag{14}$$

where the subscript t in x_t denotes the iteration number. Note that we can compute $A^T y$ only once and but need to compute $A^T A x_t$ multiple times across iterations.

One possible scenario is to encode both A and A^T separately and then use the coded matrix-vector multiplication method twice every iteration; once for Ax_t using the coding on A and once for $A^T(Ax_t)$ using the coding on A^T. Another choice is to compute A^TA offline and encode directly the product A^TA. In this case, we use the coded matrix-vector multiplication method only once per iteration.

Figure 6 compares the uncoded and the polar coded distributed computation methods along with different values for N and erasure probabilities ϵ. In this experiment we have pre-computed and encoded the matrix product A^TA. Then, in each iteration of the gradient descent, the central node decodes the downloaded outputs, updates x_t, sends the updated x_t to AWS S3 and initializes the computation A^TAx_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 central node, and there is no decoding. Then, the central node computes and sends the updated x_t, and initializes the next iteration. The data matrix has dimensions $A \in \mathbb{R}^{20000\times 4800}$, the variable is $x \in \mathbb{R}^{4800\times 1000}$, and the output is $y \in \mathbb{R}^{20000\times 1000}$. We have randomly generated the data used in this experiment.

We note that in a given iteration, while computation with polar coding with rate $(1-\epsilon)$ requires waiting for the first decodable set of outputs out of N outputs, uncoded computation waits for all $N(1-\epsilon)$ nodes to finish computation. Using ϵ as a tuning parameter for redundancy, we achieve different convergence times.

3.3 ImageNet: Large-Scale Experiment

Figure 7 shows the cost ($\|Ax-y\|_2^2$) against wall-clock time when we solve the least squares problem where the data matrix A consists of the first 128 classes of the ImageNet dataset [36]. This experiment aims to demonstrate that gradient descent with coded matrix multiplication can be used to speed up fine-tuning of pre-trained machine learning models.

Each data sample of the ImageNet dataset is an RGB picture of (rescaled) dimensions $256 \times 256 \times 3$. Figure 7 compares the computation speeds of the naive approach with no

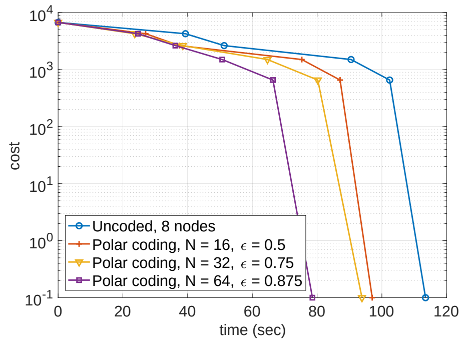


Figure 6: Cost ($\|Ax-y\|_2^2$) against wall-clock time for the gradient descent example.

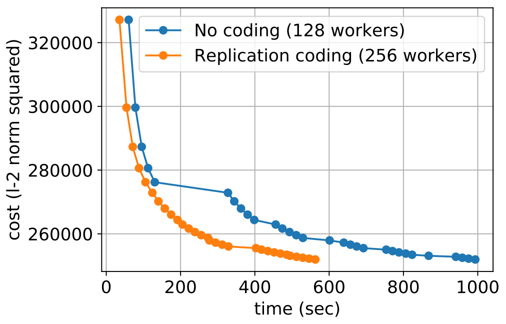


Figure 7: Large-scale gradient descent example on ImageNet.

coding (orange) and partial coded construction (blue). The circles show the beginning of each iteration. For the partial coded construction, the construction size is 2, which is equivalent to the repetition coding. Both methods have been run for 30 iterations with 5 iterations per serverless function lifetime. In other words, each serverless function has been reused for 5 iterations. Figure 7 demonstrates that coding, even for a small construction size of 2 helps speed up the computation. Larger construction sizes are expected to reduce the computation times further at the expense of increased encoding times.

We observe that the 5'th iteration for the uncoded case takes longer than 100 seconds and there are other iterations that take much longer than the rest of the iterations. This is expected when we do not have redundancy since even if there is only one straggling worker node, the central node waits for the straggling node to return its output before starting the next iteration of the gradient descent algorithm. Figure 7 verifies that this effect is mitigated with replication coding. Furthermore, we note that the coded version achieves roughly a 50% reduction in the computation time while using twice many AWS Lambda functions. Thus, the price of the overall computation stays the same since pricing is calculated based on the duration of time that the functions take executing. This experiment verifies the effectiveness of our approach as it leads to faster computing while keeping the overall price the same.

3.4 Extension to Optimizing Nonlinear Black-Box Functions

In this subsection, we present numerical results for the application of our randomized polar codes to black-box optimization problems. This is an extension of our approach to computing linear functions by considering a linear approximation to the gradient for nonlinear objective functions. The details are given in Section A. Our black-box optimization strategy only accesses the objective through function evaluations. This makes our method applicable to any arbitrary objective function. We consider the optimization of the nonlinear objective $f(\theta) = \|A\theta - b\|_1$ as a test case for the proposed black-box optimization method. Figure 8 shows the cost as a function of iterations when we use gradient descent algorithm with gradient estimates obtained by the finite differences method, the proposed coded black-box optimization method, and the structured evolution strategies method (these methods are described in Section A). For fairness in comparing these methods, we used straggler-resilient versions of the finite differences method and the structured evolution strategies method in obtaining these results. We wait for the first arriving 16 worker outputs out of the total 32 outputs to make a gradient update for the finite differences method. For the structured evolution strategies method, the perturbation directions are generated from the rows of the matrix product HD which is a 32×32 matrix, so we wait for the first arriving 16 outputs out of the 32.

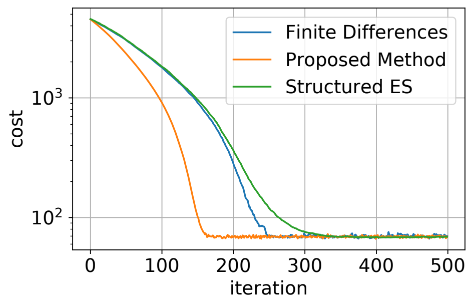
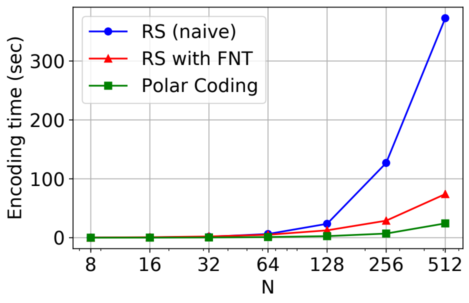


Figure 8: Black-box optimization of the function $f(\theta) = \|A\theta - b\|_1$ where $A \in \mathbb{R}^{200\times 32}, b \in \mathbb{R}^{200}$.

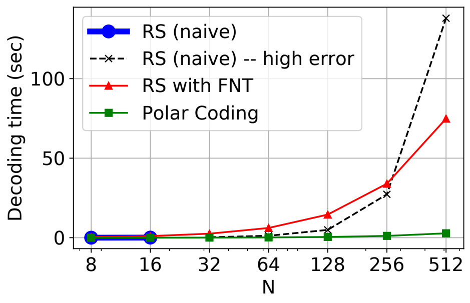
Finally, we implement the proposed method by utilizing a rate of $\frac{1}{2}$ with a total of 64 workers and wait for the first decodable set of outputs out of these 64 outputs. Figure 8 illustrates that having all the entries of the gradient estimate through decoding leads to faster convergence than having only a half of the entries of the gradient estimate. It also shows that the proposed method results in faster convergence compared to the structured evolution strategies method.

3.5 Encoding and Decoding Speed Comparison

Figure 9 shows the time that encoding and decoding algorithms take as a function of the number of nodes N for Reed-Solomon codes and our approach for exact recovery using polar codes. For Reed-Solomon codes, we have implemented two separate approaches for encoding and decoding. The first 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 Figure 9, we 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 curve in Figure 9(b) with cross markers and dashed lines, labeled as 'high error', indicates that the error due to the decoder is unacceptably high. This happens since the linear system that we need to solve for recovery is ill-conditioned.



(a) Encoding



(b) Decoding

Figure 9: Comparison of encoding and decoding speeds for RS and polar codes.

Figure 9 illustrates that our approach for exact recovery takes considerably less time for encoding and 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 Figure 9, our approach using polar codes is more suitable for serverless computing where N is large.

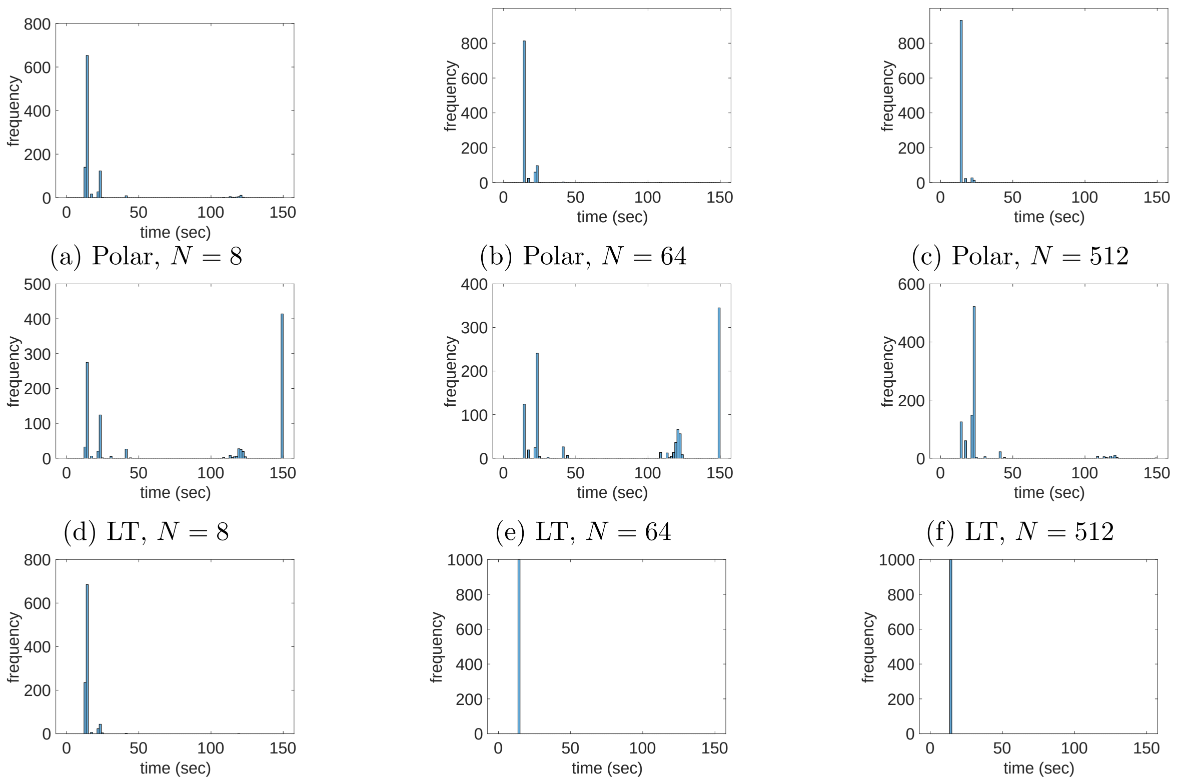
3.6 Empirical Distribution of Decodability Times

We refer to the time instance where the available outputs become decodable for the first time as *decodability time*. Figure 10 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 Figure 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 Figure 10(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 [21]. We see that polar codes achieve better decodability times than LT codes. Plots in Figure 10(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.

4 Conclusion

We have introduced a method that combines polar coding based coded computation and randomized sketching algorithms. Due to their low complexity and simple encoding and



(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, $N = 512$

Figure 10: Histograms of decodability time for polar, LT, and MDS codes.

decoding algorithms, polar codes lead to favorable run time performance in coded computation. It is critical to note that a low complexity decoder is particularly advantageous when addressing large-scale datasets and a high number of nodes.

A number of works in coded computation literature employ MDS codes for inserting redundancy into computations. In the case where one wishes to work with full-precision data and use Reed-Solomon codes, decoding requires solving a linear system. This will require cubic complexity and lead to unstable solutions for systems with high number of nodes since we are solving a Vandermonde based linear system. Furthermore, 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 [37], 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 [38] 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 large 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.

Acknowledgements

This work was supported by the National Science Foundation under grants NSF CAREER CCF-2236829, ECCS-2037304, DMS-2134248, Army Research Office Early Career Award W911NF-21-1-0242, ACCESS – AI Chip Center for Emerging Smart Systems sponsored by InnoHK funding, Hong Kong SAR, and a Precourt Pioneering Projects seed grant. We would like to thank the anonymous reviewers for their suggestions that improved our paper.

References

[1] B. Bartan and M. Pilanci, "Straggler resilient serverless computing based on polar codes," in *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2019, pp. 276–283.

[2] ——, “Distributed black-box optimization via error correcting codes,” in *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2019, pp. 246–252.

[3] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, March 2018.

[4] 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)*, pp. 1993–1997, 2018.

[5] 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*, pp. 4406–4416, 2017.

[6] T. Dean and M. Boddy, “An analysis of time-dependent planning,” in *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*, ser. AAAI’88. AAAI Press, 1988, p. 49–54.

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

[8] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, “On the optimal recovery threshold of coded matrix multiplication,” *IEEE Transactions on Information Theory*, vol. 66, no. 1, pp. 278–301, 2020.

[9] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, “Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding,” in *2018 IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 2022–2026.

[10] Q. Yu and A. S. Avestimehr, "Entangled polynomial codes for secure, private, and batch distributed matrix multiplication: Breaking the "cubic" barrier," in *2020 IEEE International Symposium on Information Theory (ISIT)*, 2020, pp. 245–250.

[11] H. Esfahanizadeh, A. Cohen, M. Médard, and S. Shamai Shitz, "Distributed computations with layered resolution," in *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, 2022, pp. 257–261.

[12] C.-S. Yang, R. Pedarsani, and A. S. Avestimehr, "Timely coded computing," in *2019 IEEE International Symposium on Information Theory (ISIT)*, 2019, pp. 2798–2802.

[13] T. Jahani-Nezhad and M. A. Maddah-Ali, "Berrut approximated coded computing: Straggler resistance beyond polynomial computing," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 1, pp. 111–122, 2023.

[14] S. Wang, J. Liu, and N. Shroff, "Fundamental limits of approximate gradient coding," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 3, dec 2019. [Online]. Available: https://doi.org/10.1145/3366700

[15] S. Wang, J. Liu, N. Shroff, and P. Yang, "Computation efficient coded linear transform," in *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and M. Sugiyama, Eds., vol. 89. PMLR, 16–18 Apr 2019, pp. 577–585. [Online]. Available: https://proceedings.mlr.press/v89/wang19a.html

[16] S. Ubaru, A. Mazumdar, and Y. Saad, "Low rank approximation and decomposition of large matrices using error correcting codes," *IEEE Trans. Inf. Theor.*, vol. 63, no. 9, p. 5544–5558, sep 2017. [Online]. Available: https://doi.org/10.1109/TIT.2017.2723898

[17] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, "Coded computation over heterogeneous clusters," *IEEE Transactions on Information Theory*, vol. 65, no. 7, pp. 4227–4242, 2019.

[18] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, “Oversketch: Approximate matrix multiplication for the cloud,” in *2018 IEEE International Conference on Big Data (Big Data)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2018, pp. 298–304. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/BigData.2018.8622139

[19] M. Pilanci, “Computational polarization: An information-theoretic method for resilient computing,” *IEEE Transactions on Information Theory*, pp. 1–1, 2021.

[20] 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.

[21] A. Mallick, M. Chaudhari, U. Sheth, G. Palanikumar, and G. Joshi, “Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication,” *SIGMETRICS Perform. Eval. Rev.*, vol. 48, no. 1, p. 95–96, jul 2020. [Online]. Available: https://doi.org/10.1145/3410048.3410104

[22] J. S. Miguel and N. E. Jerger, “The anytime automaton,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 545–557.

[23] N. S. Ferdinand and S. C. Draper, “Anytime coding for distributed computation,” in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2016, pp. 954–960.

[24] N. Ferdinand, B. Gharachorloo, and S. C. Draper, “Anytime exploitation of stragglers in synchronous stochastic gradient descent,” in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2017, pp. 141–146.

[25] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, , and R. Katz, “A case for serverless machine learning,” *Workshop on Systems for ML and Open Source Software at NeurIPS 2018*, 2018.

[26] V. Gupta, S. Phade, T. Courtade, and K. Ramchandran, "Utility-based resource allocation and pricing for serverless computing," 2020. [Online]. Available: https://arxiv.org/abs/2008.07793

[27] 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)*, July 2018, pp. 334–341.

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

[29] V. Gupta, S. Kadhe, T. Courtade, M. W. Mahoney, and K. Ramchandran, "Oversketched newton: Fast convex optimization for serverless systems," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 288–297.

[30] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[31] K. Choromanski, M. Rowland, V. Sindhwani, R. Turner, and A. Weller, "Structured evolution with compact architectures for scalable policy optimization," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 970–978. [Online]. Available: https://proceedings.mlr.press/v80/choromanski18a.html

[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[33] M. W. Mahoney, "Randomized algorithms for matrices and data," *Foundations and Trends in Machine Learning in Machine Learning*, vol. 3, no. 2, 2011.

[34] J. A. Tropp, "Improved analysis of the subsampled randomized hadamard transform," *Advances in Adaptive Data Analysis*, vol. 3, no. 01n02, pp. 115–126, 2011.

[35] F. Krahmer and R. Ward, "New and improved johnson–lindenstrauss embeddings via the restricted isometry property," *SIAM Journal on Mathematical Analysis*, vol. 43, no. 3, pp. 1269–1281, 2011.

[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

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

[38] F. Didier, "Efficient erasure decoding of reed-solomon codes," *CoRR*, vol. abs/0901.1886, 2009. [Online]. Available: http://arxiv.org/abs/0901.1886

[39] S. Boyd and L. Vandenberghe, *Convex optimization.* Cambridge, UK: Cambridge University Press, 2004.

A Extension to Coded Black-Box Optimization

In this section, we extend our coding method to the minimization of $f(\theta)$ where $f : \mathbb{R}^d \to \mathbb{R}$. We assume that we do not have access to the analytical form of $f(\theta)$ or its gradient, and we assume that we are only able to make queries for function evaluations. This setting is called black-box optimization.

We use ∇_v to denote directional derivative along the direction $v \in \mathbb{R}^d$. We use e_i to denote the i'th unit column vector with the appropriate dimension. H refers to the Hadamard matrix (its dimension can be determined from the context), and $\mathbf{H}$ is the Hessian of a function. We now briefly describe the black-box optimization methods of finite differences and evolution strategies and then present the proposed coded black-box optimization method.

Finite Differences: For black-box optimization problems, approximate gradients can be obtained by using the finite differences estimator. The derivative of a function $f(x)$ with respect to the variable x_i can be approximated by using

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x+\delta e_i) - f(x-\delta e_i)}{2\delta}, \tag{15}$$

where $\delta \in \mathbb{R}$ is a small scalar that determines the perturbation amount. One can obtain an approximate gradient using (15) and use it in gradient-based optimization methods such as gradient descent.

The finite differences estimator can easily exploit parallelism since each partial approximate derivative can be independently evaluated in different worker nodes in parallel. This method can be made straggler-resilient by using only the available derivative estimates and ignoring the outputs of the slower workers. However, this typically leads to slower convergence. Another alternative is to replicate finite difference calculations, which is not optimal from a coding theory perspective.

Evolution Strategies: Let us consider the following evolution strategies (ES) gradient estimator ([30], [31])

$$\nabla f(\theta) \approx \frac{1}{2N\delta} \sum_{i=1}^{N} (f(\theta + \delta\epsilon_i)\epsilon_i - f(\theta - \delta\epsilon_i)\epsilon_i)\,, \tag{16}$$

where δ is the scaling coefficient for the random perturbation directions ϵ_i and N is the number of perturbations. This estimator is referred to as antithetic evolution strategies gradient estimator. The random perturbation directions ϵ_i may be sampled from a standard multivariate Gaussian distribution $\mathcal{N}(0, I)$. Alternatively, ϵ_i's may be generated as the rows of the matrix HD where H is the Hadamard matrix and D is a diagonal matrix with entries distributed as Rademacher distribution.

It is shown in [31] that if exploration (or perturbation) directions ϵ_i are orthogonal, the gradient estimators lead to a lower error. We note that the rows of HD are orthogonal with the appropriate scaling factor. We omit this scaling factor by absorbing it in the δ term which scales perturbation directions ϵ_i. This work considers the case where random perturbation directions are generated according to HD and we will refer to it as *structured evolution strategies* as it is done in [31].

Evolution strategies can exploit parallelism as well since workers need to communicate only scalars which are the function evaluations and the random seeds used when generating the random perturbation directions.

A.1 Distributed Black-Box Optimization using Polar Codes

We now present the proposed method for speeding up distributed black-box optimization. We start by introducing some more notation and definitions. The derivative of a differentiable function f at a point θ along the unit vector direction e_i is the i'th component of the gradient ∇f, that is, $\frac{\partial f}{\partial \theta_i} = e_i^T \nabla f$. The directional derivative along v is defined as follows

$$\nabla_v f = \lim_{\delta \to 0} \frac{f(\theta + \delta v) - f(\theta)}{\delta}\,. \tag{17}$$

If the function is differentiable at a point θ, then the directional derivative exists along any direction v and is a linear map [39]. In this case we have

$$\nabla_v f = v^T \nabla f. \tag{18}$$

When we do not have access to exact gradients, we can employ a *numerical directional derivative* by choosing a small δ in

$$\nabla_v f \approx \frac{f(\theta + \delta v) - f(\theta)}{\delta}\,. \tag{19}$$

Note that the approximation in (19) is not symmetric, that is, it involves perturbing the parameters only along $+v$. We instead use the symmetric version of (19) for approximating derivatives in which the parameters are perturbed along both the directions $-v$ and $+v$:

$$\nabla_v f \approx \frac{f(\theta + \delta v) - f(\theta - \delta v)}{2\delta}\,. \tag{20}$$

If we consider the Taylor series expansion for $f(\theta + \delta v)$ and $f(\theta - \delta v)$, we obtain

$$\begin{aligned} f(\theta + \delta v) &= f(\theta) + \delta \nabla f^T v + \frac{\delta^2}{2} v^T \mathbf{H} v + O(\delta^3) \\ f(\theta - \delta v) &= f(\theta) - \delta \nabla f^T v + \frac{\delta^2}{2} v^T \mathbf{H} v + O(\delta^3) \end{aligned} \tag{21}$$

where $O(\delta^3)$ is a third order error term and $\mathbf{H}$ is the Hessian matrix for f. Substituting these expansions in (20), we obtain

$$\nabla_v f \approx \frac{2\delta \nabla f^T v + O(\delta^3)}{2\delta} = \nabla f^T v + O(\delta^2)\,. \tag{22}$$

This shows that for small δ, the numerical directional derivative becomes approximately linear in ∇f. Our proposed method makes use of this assumption that directional derivative estimates are approximately linear in the directions v to ensure that coding can be applied to

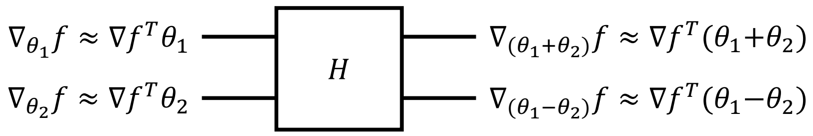

$\nabla_{\theta_1} f \approx \nabla f^T \theta_1$
$\nabla_{\theta_2} f \approx \nabla f^T \theta_2$
H
$\nabla_{(\theta_1+\theta_2)} f \approx \nabla f^T (\theta_1+\theta_2)$
$\nabla_{(\theta_1-\theta_2)} f \approx \nabla f^T (\theta_1-\theta_2)$

Figure 11: 2-by-2 construction based on Hadamard transformation.

directional derivative estimates. To make this more concrete, let us consider the construction given in Figure 11. The block H corresponds to the Hadamard kernel $H = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$.

In Figure 11, if we know the estimates for $\nabla_{(\theta_1+\theta_2)} f$ and $\nabla_{(\theta_1-\theta_2)} f$, we can compute the estimates for $\nabla_{\theta_1} f$ and $\nabla_{\theta_2} f$ because the directional derivative estimates are approximately linear in their corresponding directions. Furthermore, if we know the estimate for $\nabla_{\theta_1} f$, then it is sufficient to know only one of the estimates for $\nabla_{(\theta_1+\theta_2)} f$ or $\nabla_{(\theta_1-\theta_2)} f$ in order to be able to compute the estimate for $\nabla_{\theta_2} f$. This would happen if, for example, θ_1 is the zero vector (i.e. frozen direction) because the estimate for $\nabla_{\theta_1} f$ would be zero and either of the directional derivative estimates from the right-hand side would be enough for us to obtain the estimate of $\nabla_{\theta_2} f$. We refer to the directions θ_1, θ_2, $(\theta_1 + \theta_2)$, $(\theta_1 - \theta_2)$ as perturbation directions.

We now summarize the proposed method and the descriptions of the steps will follow.

- Encode all the unit vectors in $\mathbb{R}^d$ to obtain the encoded perturbation directions.
- Assign each perturbation direction to a worker node and have them compute their directional derivative estimates using (20).
- Central node starts collecting worker outputs.
- When a decodable set of worker outputs is available, the central node decodes these outputs to obtain an estimate for the gradient.
- The central node computes the next iterate for the parameter θ.
- Repeat until convergence or for a desired number of iterations.

The above procedure assumes that we want to estimate all entries of the gradient, but it is possible estimate only a portion of gradient entries by encoding only the unit vectors

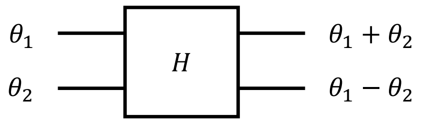


Figure 12: 2-by-2 Hadamard transformation of the perturbation directions.

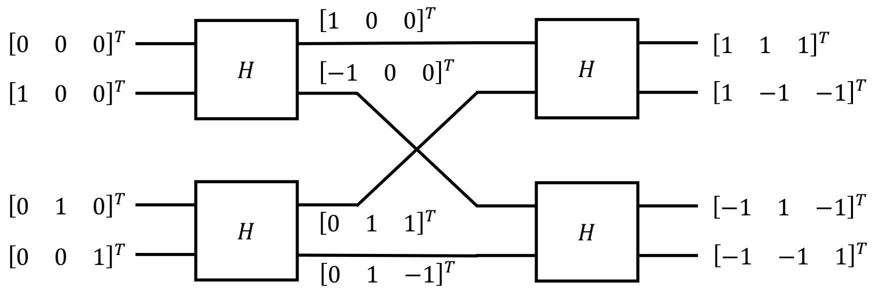


Figure 13: Example encoding for $N = 4$

corresponding to the desired entries. Moreover, we note that we can always check whether decoding helps in obtaining a better objective function compared to the the structured evolution strategies and make the update accordingly. Since the decoding step is fast due to the efficient polar decoder, decoding the outputs but not using the recovered estimate does not place a heavy computational burden.

A.2 Encoding

Encoding is computed based on the Hadamard transformation whose kernel is shown in Figure 12. In channel coding, freezing channels corresponds to sending known bits, e.g., the zero bit. Here, freezing inputs corresponds to setting them to all-zero coordinates so that the corresponding directional derivative is zero. This makes it possible, when decoding, to take the value of the derivative estimates for frozen nodes to be zero. For the information nodes, we simply send in unit vectors. For instance, encoding for a function of 3 variables using $N = 4$ workers is shown in Figure 13. The resulting 4 output vectors are the perturbation directions.

In this construction, the rate is 3/4 since 3 out of 4 inputs are used for sending in unit vectors. When the erasure probabilities of the transformed nodes are computed (see [7]), one will see that the worst one corresponds to the first index. Hence, the first input is frozen and the remaining 3 inputs are the information nodes. The frozen input is set to the zero vector and the others are set to 3-dimensional unit vectors. Note that for the frozen input, perturbing by the zero vector is the same as not perturbing the variables and hence we get $f(\theta+0) - f(\theta-0) = 0$. This is important since during decoding, we will not have to do any computations to evaluate the value of the frozen inputs as we know they are zero.

If we wish to use $N = 8$ workers instead of 4, we would set the worst 5 inputs to zero vectors and the remaining best three inputs would be set to the unit vectors e_1, e_2, e_3. The rate in this case would be 3/8, and this construction would be more straggler-resilient since we can recover the gradient estimate in the presence of even more stragglers compared to the $N = 4$ case.

Embedding interpretation: It is also possible to perform the encoding step slightly differently for a different view on freezing channels. Instead of setting frozen inputs to zero vectors, one can increase the dimension of the inputs from d to N and set the frozen channels to unit vectors e_j where $j \in \{d+1, d+2, \ldots, N\}$. Because the function $f(\theta)$ accepts d-dimensional inputs, we could embed $f(\theta)$ into a higher dimension, that is, we could define $\tilde{f}(\tilde{\theta})$ where $\tilde{\theta} \in \mathbb{R}^N$ and $\tilde{f}(\tilde{\theta}) = f(\theta)$ if $\tilde{\theta}_i = \theta_i$ for $i \in \{1, 2, \ldots, d\}$. The advantage of this approach is that the output of the encoding step will be equal to the H matrix with permuted rows.

A.3 Decoding

The sequential decoder given in Algorithm 2 directly handles linear operations such as matrix-vector multiplication with full-precision data (i.e. does not require finite field data). Since the gradient estimates $\nabla_{h_i} f$ can be linearly approximated, we can use the same decoding method.

We note that the structured evolution strategies method is based on exploring the parameter space along the rows of HD instead of H alone. So far, we have only considered the H matrix for perturbation directions. It is possible to incorporate the diagonal matrix D into our method as well. Multiplying H by D from the right corresponds to multiplying all the entries of the i'th column of H by D_i for all i. In this case, instead of computing an estimate for the directional derivative along a direction of v, we approximate the directional derivative with respect to the direction Dv.

B Proofs

Proof of Lemma 2.7. 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 2.6. Since f satisfies the linearity property, we can write

$$\begin{bmatrix} f(v_1) \\ f(v_2) \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \times \begin{bmatrix} f(u_1) \\ f(u_2) \end{bmatrix}. \tag{23}$$

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

$$K_{12} \times f(u_2) = f(v_1) - K_{11} f(u_1) \tag{24}$$

$$K_{22} \times f(u_2) = f(v_2) - K_{21} f(u_1). \tag{25}$$

We use (24) if $f(v_1)$ is known, and (25) if $f(v_2)$ is known. This implies that both K_{12} and K_{22} need to be non-zero. 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 non-zero and its second element as zero, or K must be invertible. Since we already found out that K_{12} and K_{22} are both non-zero, 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 non-zero. 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 (24), (25) because K_{12} and K_{22} are both assumed to be non-zero. Hence this completes the proof that a kernel K satisfying the given two conditions is a polarizing kernel. $\square$

Proof of Theorem 2.8. By Lemma 2.7, we know that for K to be a polarizing kernel, it must be invertible and must have both K_{12} and K_{22} as non-zero. For a 2×2 matrix to be invertible with both second column elements as non-zero, at least one of the elements in the first column must also be non-zero. We now know that K_{12}, K_{22} and at least one of K_{11}, K_{21} must be non-zero in a polarizing kernel K. It is easy to see that having all four elements of K as non-zero leads to more computations than having only three elements of K as non-zero. 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 non-zero 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 non-zero elements. $\square$

C Partial Construction

In this section, we introduce a novel idea that we refer to as *partial construction* to scale up the encoding procedure. Suppose that we are interested in computing the linear operation Ax where we only encode A and not x. If the data matrix A is extremely large, it may be time consuming to encode the data. A way around having to encode a large A is to consider partial code constructions, that is, for $A = [A_1^T, \ldots, A_p^T]^T$, we encode each submatrix A_i separately. The encoded A can be written as follows:

$$\text{encode}(A) = \begin{bmatrix} (Z \otimes I_{n/(ps)})A_1 \\ \vdots \\ (Z \otimes I_{n/(ps)})A_p \end{bmatrix}. \tag{26}$$

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 amount of 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 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 infeasible, one can construct many smaller size codes. When the code size is small enough, a Vandermonde-based linear system can be painlessly solved.

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 is changing over time because it might be too expensive to encode the entire dataset A every time it gets updated.

D 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 $A = [A_1^T, \ldots, A_{d_1}^T]^T$ and $B = [B_1, \ldots, B_{d_2}]$. Let us denote zero matrix padded version of A by $\tilde{A} = [\tilde{A}_1^T, \ldots, \tilde{A}_{N_1}^T]^T$ 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 $\tilde{B} = [\tilde{B}_1, \ldots, \tilde{B}_{N_2}]$ 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:

$$P = \begin{bmatrix} (G_{N_1}\tilde{A})_1(\tilde{B}G_{N_2})_1 & \ldots & (G_{N_1}\tilde{A})_1(\tilde{B}G_{N_2})_{N_2} \\ \vdots & \ddots & \\ (G_{N_1}\tilde{A})_{N_1}(\tilde{B}G_{N_2})_1 & \ldots & (G_{N_1}\tilde{A})_{N_1}(\tilde{B}G_{N_2})_{N_2} \end{bmatrix}. \tag{27}$$

Note that for fixed j, the worker outputs are:

$$\begin{bmatrix} (G_{N_1}\tilde{A})_1(\tilde{B}G_{N_2})_j \\ \vdots \\ (G_{N_1}\tilde{A})_{N_1}(\tilde{B}G_{N_2})_j \end{bmatrix}. \tag{28}$$

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:

$$\begin{bmatrix} (G_{N_1}\tilde{A})_i(\tilde{B}G_{N_2})_1 & \dots & (G_{N_1}\tilde{A})_i(\tilde{B}G_{N_2})_{N_2} \end{bmatrix}. \tag{29}$$

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, the decoder algorithm for the 2D case is given in Algorithm 4. 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 (27). 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 4: 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. 2
            forward propagation 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. 2
            forward propagation to fill in $P[:,j]$
decode all rows and then all columns of $P$
return entries of $P$ (ignoring the frozen entries)
```

E Privacy

Worker nodes may be unreliable in distributed computing and in such cases it is desirable to introduce privacy into the computation. It is possible to incorporate privacy into coded computation with polar codes in a very straightforward way. For simplicity, let us assume that no two workers can collude. Then, we note that selecting the last input as a random matrix R in the code construction, that is, $U_N = R$, leads to all worker outputs containing the product Rx as an additive term. Hence, this results in a privacy-preserving computation against single-node attacks.