Local Re-encoding for Coded Matrix Multiplication

Xian Su, *Student Member, IEEE,* Jared Parker, Xiaomei Zhong, Xiaodi Fan, Jun Li, *Member, IEEE*

Matrix multiplication is a fundamental operation in various algorithms for big data analytics and machine learning. As the size of the dataset increases rapidly, it is now a common practice to distribute the computation on multiple servers. As straggling servers are inevitable in a distributed infrastructure, various coding schemes have been proposed to tolerate potential stragglers. However, as resources are shared with other jobs in a distributed infrastructure and their performance can change dynamically, the optimal way of encoding the input matrices is not static. So far, all existing coding schemes require encoding input matrices in advance, and cannot change the coding schemes or adjust their parameters flexibly. In this paper, we propose a framework that can change the coding schemes and/or their parameters by locally re-encoding the coded task on each server. We first present this framework for entangled polynomial codes, which changes the coding parameters with marginal overhead and saves job completion time. We then extend the framework for matrices with bounded entries, achieving a higher level of flexibility for local re-encoding while maintaining better numerical stability.

***Index Terms*—big data infrastructure, matrix multiplication, re-encoding, numerical stability**

I. Introduction

MATRIX multiplication is an essential building block in various algorithms for big data analytics and machine learning. With the growing sizes of the dataset and the model, it is now common that the input matrices are so large that their multiplication cannot be computed on a single server. Therefore, it becomes inevitable to run the multiplication on multiple servers in a distributed infrastructure, *e.g.*, in a cloud, where each server executes a task multiplying two smaller submatrices. However, it is well known that servers in a distributed infrastructure may experience temporary performance degradation, due to load imbalance or resource congestion [2]–[5]. Therefore, when distributing computation on multiple servers, the progress of the multiplication can be significantly affected by the tasks running on such slow or failed servers, which we call *stragglers*.

In order to tolerate stragglers in distributed matrix multiplication, a naive method is to replicate each task on multiple servers. For example, with two input matrices $A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$ and $B = \begin{bmatrix} B_0 & B_1 \end{bmatrix}$, we have $AB = \begin{bmatrix} A_0B_0 & A_0B_1 \\ A_1B_0 & A_1B_1 \end{bmatrix}$, or $AB = [A_iB_j]_{2\times 2}$ for simplicity. Then we split the job into four tasks A_iB_j, $i \in [0,1]$, $j \in [0,1]$, and replicate each of such four tasks on multiple servers. This method, however, requires a large number of tasks to tolerate just a small number of stragglers. To tolerate only r stragglers, we need to replicate all tasks $r+1$ times.

X. Su and X. Fan are with the Graduate Center of the City University of New York. J. Parker is with the Department of Computer Science, Virginia Commonwealth University. X. Zhong is with the School of Software Engineering, East China Jiaotong University. J. Li is with the Queens College and the Graduate Center of the City University of New York.

This paper was presented in part at the 2020 IEEE International Symposium on Information Theory [1].

On the other hand, coding-based approaches for distributed matrix multiplication have been proposed to tolerate stragglers more efficiently [3]–[10], where each server multiplies coded matrices encoded from submatrices in A or/and B. Using the same example above, we still partition A and B into two submatrices, and encode them into $(A_0 + A_1)$ and $(B_0 + B_1)$, respectively. Then an additional coded task can be created to compute $(A_0 + A_1)(B_0 + B_1)$. Since $(A_0 + A_1)(B_0 + B_1) = A_0B_0 + A_0B_1 + A_1B_0 + A_1B_1$, a sum of the result of the four original tasks, we can recover the four submatrices in AB once we have the results of any four of the five tasks. Compared with replicating each task on two servers, this coding scheme can tolerate any single straggler with 75% fewer additional tasks.

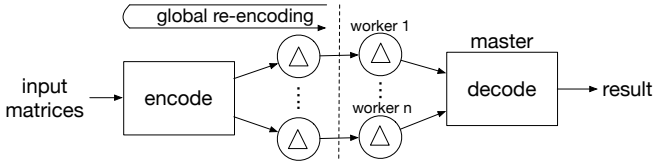


Fig. 1: The model of coded matrix multiplication with global re-encoding.

Conceptually, we illustrate the workflow of conventional coded matrix multiplication in Fig. 1. To create additional coded tasks, the original input matrices need to be partitioned and encoded, and such coded tasks are initially stored on n servers, which we call *workers*. Each worker returns the result of the multiplication of its coded matrices to a *master* server. With a carefully designed coding scheme, the master can decode from a subset of the n workers without waiting for the stragglers. However, since it takes time to generate all coded matrices, and the execution of tasks may not start immediately after encoding (*e.g.*, due to the task scheduling), the coding

scheme and its parameters chosen before encoding may not be the best choice when the tasks are being executed, as the performances of resources in a distributed infrastructure are subject to change due to the shared nature of resources in the distributed infrastructure, especially in the cloud. To the best of our knowledge, we can only change the coding scheme or its parameters by *global re-encoding*, *i.e.*, re-encoding coded tasks from scratch. In the global re-encoding, it is necessary to read the input matrices, split them into submatrices in a different way, encode them into coded tasks with a different coding scheme or with different values of parameters, and finally store them on workers. Inevitably, it consumes a significant amount of time and network bandwidth to encode and distribute new coded matrices.

In this paper, we propose a framework for distributed matrix multiplication that supports changing the coding schemes and/or the values of their parameters by *locally* re-encoding the coded matrices on each server, without receiving any additional data from any other server. Local re-encoding makes it possible to determine the coding scheme and/or its parameters almost immediately before the start of the computation, as shown in Fig. 2, thanks to its very low overhead. In other words, local re-encoding also gives a second chance to choose the coding scheme and/or its parameters after the initial encoding, especially when coded tasks will not run immediately after encoding. We first propose a framework of local re-encoding for entangled polynomial codes [7]. It not only supports changing the values of parameters, but also allows changing the coding schemes from the other two representative codes for matrix multiplication, *i.e.*, polynomial codes [8] and MatDot codes [9], to entangled polynomial codes.

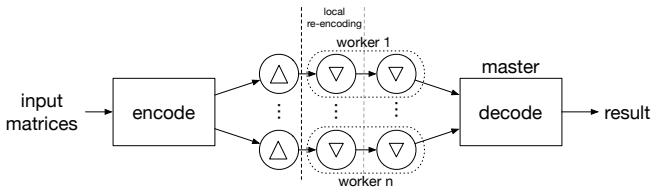


Fig. 2: The model of coded matrix multiplication with local re-encoding.

Besides local re-encoding for entangled polynomial codes, we further extend the original framework to mitigate the issue of numerical stability. As all the three coding schemes above are constructed as polynomials (elaborated in Sec. V-A), they are known to suffer from poor numerical stability for the multiplication of matrices with real numbers. The decoded result may be significantly different from the actual result due to even small perturbations during decoding, caused by the imprecision of floating-point numbers in the computer. This is because generator matrices of these three coding schemes are all Vandermonde matrices, known as numerically unstable with real numbers [11], [12].[1] Then the numerical error after local re-encoding may become significantly higher than the original one. Hence, we extend our framework to support a variation of the entangled polynomial code for matrices with bounded entries, which maintains better numerical stability [13]. Meanwhile, a more flexible tradeoff between numerical stability and computational overhead can be achieved in the extended framework.

II. Related Work

In order to tolerate stragglers in a job of distributed computing, a common method is to relaunch the affected tasks on a replacement server after the straggler is detected [14], [15]. While relaunching a task on another server may still consume a significant amount of time, we can launch each task in the job multiple times on different servers at the beginning of the job [8], [14], [16]–[19]. In this way, only one of the replicas will be sufficient to complete each task. However, replication-based methods suffer from high resource consumption. On the other hand, existing literature has proposed a number of coding-based methods that can tolerate the same number of stragglers by adding fewer additional tasks than replication-based methods. In the first effort made by Lee *et al.* [3], a coding scheme based on MDS codes was proposed for the matrix-vector multiplication, where the matrix is horizontally split and encoded to create coded tasks. More coding schemes, such as sparse coding [6] and rateless coding [20], were also proposed for the matrix-vector multiplication.

As for matrix-matrix multiplication, the coding schemes will need to split and encode two input matrices, as both of them may be large. Coding schemes based on product codes create the coded tasks in two steps [21]–[24]. A job is firstly encoded into intermediate coded tasks by applying one coding scheme to one matrix, and each intermediate task is encoded again by applying another (or the same) coding scheme on the other matrix. However, since the actual tasks are created from intermediate coded tasks, the patterns of stragglers tolerable are limited as each intermediate tasks need to be decodable. On the other hand, polynomial codes [8] and MatDot [9] codes can directly encode the two input matrices while tolerating any patterns of stragglers, as long as the number of stragglers is less than a given number. However, each input matrix can only be split in one dimension, either vertically or horizontally in such two coding schemes. More generally, entangled polynomial codes [7] and PolyDot [9] codes support to split the two input matrices vertically and horizontally at the same time.

[1]This issue does not exist when the multiplication is performed on matrices on a finite field. We focus on the multiplication of matrices of real numbers in this paper.

Many existing coding schemes for distributed matrix multiplication are constructed based on polynomials [3], [7], [8]. Two input matrices A and B are encoded as the evaluations of two polynomial(s) of $\tilde{A}(x)$ and/or $\tilde{B}(x)$, where the value of x in each task must be unique. By carefully designing the polynomial, decoding can be done by interpolating the coefficients of $\tilde{C}(x) = \tilde{A}(x)\tilde{B}(x)$, such that all submatrices of AB appear in the coefficients of $\tilde{C}(x)$. However, given multi-point evaluations of $\tilde{C}(x)$, the interpolation is equivalent to solving a linear system with a Vandermonde matrix, which is known to have a large condition number, *i.e.*, a small perturbation in the Vandermonde matrix, which is inevitable due to the limited precision of floating point numbers in the computer, may lead to a large error after decoding [12]. Hence, numerically stable codes have been proposed to achieve the same level of tolerance against stragglers with much lower errors after decoding [11], [13].

Conventionally the computation is considered as the major bottleneck in distributed systems. In order to make use of more resources efficiently, the coding schemes that leverage partial stragglers have also been proposed recently [25], [26]. On the other hand, other resources such as the bandwidth may also become a bottleneck, and communication-efficient coding schemes for distributed matrix multiplication have been proposed, *e.g.*, squeezed polynomial codes that introduce replicated coded matrices [10] and local error-correcting codes in serverless systems [5]. However, the existing works above have only considered one kind of resource with a fixed coding scheme and parameters, while in practice the performances of different resources may vary with time.

The tradeoff between different resources makes the choice of coding schemes and the values of parameters more difficult. Moreover, the optimal choice may change dynamically with time in many cases, so the optimal choice of the coding scheme and its parameters also keeps changing dynamically. To address this problem, re-encoding was proposed in distributed storage systems, where Maturana and Rashmi proposed *convertible codes* allowing the change of the parameters of MDS codes with the optimal overhead of data transfer [27], [28]. However, it only works in the distributed storage systems without considering the computation on such data. Even worse, it requires obtaining additional data from remote workers.

In this paper, we, for the first time, propose a coding framework that supports the local re-encoding of tasks for distributed matrix multiplication, which allows changing the coding scheme or/and its parameters without data transfer. This feature gives a second chance to choose the coding scheme and/or its parameters just before the start of the job. Moreover, we further extend our framework to support maintaining numerical stability. By local re-encoding with low overhead, flexible tradeoffs can be easily achieved, such as that between computation and communication, and that between numerical stability and job completion time.

III. Background and Examples

A. Background: Coding Schemes for Distributed Matrix Multiplication

In this paper, we demonstrate that coded tasks with polynomial codes [8] or MatDot codes [9] can be locally re-encoded into those with entangled polynomial codes [7], or be updated with different values of their parameters. We will first present the preliminary knowledge about such three coding schemes, and our framework in the rest of this paper will be based on this background knowledge. We assume that coded tasks are created for the multiplication of two large matrices A and B, *i.e.*, AB.

Polynomial codes assume that the input matrices A and B can be horizontally and vertically split into m and n submatrices, respectively. In other words, $A = \begin{bmatrix} A_0^T & \cdots & A_{m-1}^T \end{bmatrix}^T$ and $B = \begin{bmatrix} B_0 & \cdots & B_{n-1} \end{bmatrix}$. Hence, $AB = [A_i B_j]_{m \times n}$, and the result of the multiplication can be obtained if we can have the mn submatrices in AB. An (m, n) polynomial code encodes A and B as two polynomial functions of x, *i.e.*, $\tilde{A}_{\text{P}}(x) = \sum_{i=0}^{m-1} A_i x^{ni}$ and $\tilde{B}_{\text{P}}(x) = \sum_{j=0}^{n-1} B_j x^j$, respectively. A coded task will then multiply $\tilde{A}_{\text{P}}(x)$ with $\tilde{B}_{\text{P}}(x)$. Note that the values of x must be different among all coded tasks. Hence, we have

$$\tilde{A}_{\text{P}}(x)\tilde{B}_{\text{P}}(x) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} A_i B_j x^{ni+j}, \tag{1}$$

which is also a polynomial function of x with degree $mn - 1$, where $A_i B_j$, $i \in [0, m-1], j \in [0, n-1]$, appears as the coefficient of x^{ni+j}. In other words, if we run tasks of $\tilde{A}_{\text{P}}(x)\tilde{B}_{\text{P}}(x)$ with different values of x on multiple servers, we can recover all the coefficients from the results of any mn tasks by polynomial interpolation or Reed-Solomon decoding. Hence, its recovery threshold, *i.e.*, the number of tasks that is sufficient for decoding regardless of their patterns, is mn.

MatDot codes assume that A and B are split vertically and horizontally into p submatrices, respectively. In other words, $A = \begin{bmatrix} A_0 & \cdots & A_{p-1} \end{bmatrix}$ and $B = \begin{bmatrix} B_0^T & \cdots & B_{p-1}^T \end{bmatrix}^T$. Then $AB = \sum_{k=0}^{p-1} A_k B_k$. A MatDot code also encodes A and B as two polynomial functions of x, *i.e.*, $\tilde{A}_{\text{MD}}(x) = \sum_{k=0}^{p-1} A_k x^k$ and $\tilde{B}_{\text{MD}}(x) = \sum_{k=0}^{p-1} B_{p-1-k} x^k$, respectively. Still, each coded task multiplies such two polynomials with a unique value of x. We

can see that $\tilde{A}_{\text{MD}}(x)\tilde{B}_{\text{MD}}(x)$ is a polynomial function of x with degree $2p-2$, *i.e.*,

$$\tilde{A}_{\text{MD}}(x)\tilde{B}_{\text{MD}}(x)=\sum_{t=0}^{2p-2}\left(\sum_{l=\max\{0,t-p+1\}}^{\min\{p-1,t\}} A_l B_{p-1-t+l}\right)x^t. \quad (2)$$

In general, from the coefficients in (2), we can see that when $t=p-1$, $\sum_{k=0}^{p-1} A_k B_k$ appears as the coefficient of x^{p-1}. Hence, after interpolating the coefficients of $\tilde{A}_{\text{MD}}(x)\tilde{B}_{\text{MD}}(x)$, we will be able to obtain the results of AB. Since the degree of $\tilde{A}_{\text{MD}}(x)\tilde{B}_{\text{MD}}(x)$ is $2p-2$, the recovery threshold of the MatDot code is $2p-1$.

Allowing more general partitioning of A and B than polynomial codes and MatDot codes, entangled polynomial codes can further reduce the complexity of coded tasks with a more flexible choice of the recovery threshold. An (m,n,p) entangled polynomial code assumes that A and B are split both vertically and horizontally into $m\times p$ and $p\times n$ submatrices, respectively. In other words, $A=[A_{i,j}]_{m\times p}$ and $B=[B_{i,j}]_{p\times n}$. With an entangled polynomial code, each server runs a task that calculates $\tilde{A}_{\text{EP}}(x)\tilde{B}_{\text{EP}}(x)$, where $\tilde{A}_{\text{EP}}(x)=\sum_{i=0}^{m-1}\sum_{k=0}^{p-1} A_{i,k}x^{pni+k}$, and $\tilde{B}_{\text{EP}}(x)=\sum_{j=0}^{n-1}\sum_{k=0}^{p-1} B_{p-1-k,j}x^{pj+k}$. Then we have

$$\tilde{A}_{\text{EP}}(x)\tilde{B}_{\text{EP}}(x)=\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\sum_{s=0}^{2p-2}\left(\sum_{l=\max\{0,s-p+1\}}^{\min\{p-1,s\}} A_{i,l}B_{p-1-t+l,j}\right)x^{pni+pj+s}. \quad (3)$$

Still, $\tilde{A}_{\text{EP}}(x)\tilde{B}_{\text{EP}}(x)$ is a polynomial function of x whose degree is $mnp+p-2$. Hence, we can interpolate its coefficients with any $mnp+p-1$ tasks that have different values of x. In particular, given $t\in[0,mnp+p-2]$, it can be uniquely written as $t=pni+pj+s$, where $i\in[0,m-1]$, $j\in[0,n-1]$, and $s\in[0,2p-2]$. If $s=p-1$, the coefficient of x^t is $\sum_{k=0}^{p-1} A_{i,k}B_{k,j}$. Thus, we can obtain the mn submatrices in AB from the coefficients of $\tilde{A}_{\text{EP}}(x)\tilde{B}_{\text{EP}}(x)$, and the recovery threshold of the entangled polynomial code is $mnp+p-1$.

By comparing (3) with (1) and (2), we can see that polynomial codes and MatDot codes can be considered as special cases of entangled polynomial codes. When $p=1$, the corresponding (m,n,p) entangled polynomial code becomes an (m,n) polynomial code whose recovery threshold is $mn+1-1=mn$. When $m=n=1$, it becomes a MatDot code whose recovery threshold is $p+p-1=2p-1$.

B. Motivating Examples

In order to save the time of coded distributed matrix multiplication, it seems that we can split the input matrices into partitions as many as possible, *i.e.*, increasing the values of m, n, and p. At the same time, however, the communication overhead will also increase, as more results need to be sent to the master with the increasing of the corresponding recovery threshold.

In Fig. 3, we illustrate tradeoff curves between the computation overhead and communication overhead in a job of coded matrix multiplication. We assume that the sizes of the two input matrices are both 4096×4096, whose entries are double-precision floating-point numbers. We measure the computation overhead by the number of multiplication operations in each coded task, and measure the communication overhead by the number of bytes received by the master. In Fig. 3, we demonstrate the computation and communication overheads of three series of coding schemes: MatDot codes with $p=2,3,4$, as well as $(2,1,p)$ and $(2,2,p)$ entangled polynomial codes with $p=1,2,3,4$, respectively. Specifically, MatDot codes can also be considered as $(1,1,p)$ entangled polynomial codes. When $p=1$, the two series of entangled polynomial codes can also be considered as $(2,1)$ and $(2,2)$ polynomial codes.

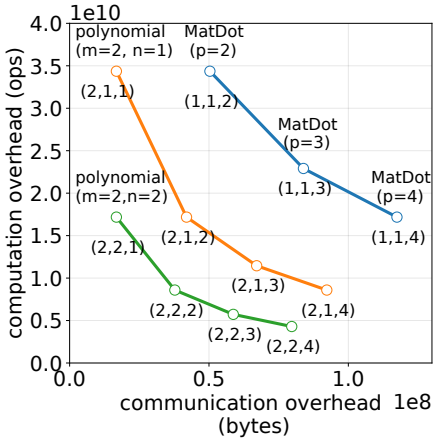


Fig. 3: The tradeoff curves between computation and communication overhead.

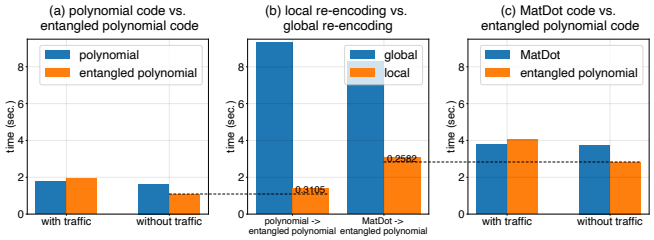


Fig. 4: Comparisons of completion time of coded distributed matrix multiplication with and without additional traffic.

From Fig. 3, we can see that in each of the three series, as p increases, the computation overhead of each task becomes less while the communication overhead increases. As the performance of servers in a distributed infrastructure can be affected by a number of factors, it is typical to consider

the performance of different resources when choosing the way in which the tasks are encoded for distributed matrix multiplication. For example, if CPU is the bottleneck, it is then desirable to split A and B into more submatrices, *i.e.*, making the recovery threshold higher in exchange for a lower complexity of each task. On the other hand, if the network bandwidth is limited, it becomes desirable to split A and B into fewer submatrices, making it possible to complete the computation on fewer tasks with lower communication overhead.

Furthermore, to demonstrate the impact of different resources on the performance of distributed matrix multiplication, we run a job that multiplies the same two 4096×4096 matrices in our local cluster. The job is implemented with Open MPI [29]. We use a master server in the cluster with the same configuration as workers, which calculates the result of coded tasks. Each worker uploads the result of its task to the master. The number of workers in each job equals the corresponding recovery threshold of the coding scheme plus the number of stragglers to tolerate, and we set to tolerate 5 stragglers. When the number of results received by the master reaches the recovery threshold, the master will stop receiving any new result and decode such results. Hence, job completion time includes the time of executing tasks on workers, uploading the results to the master, and decoding the results on the master.

In our experiment, we observe how the performance of the job, in terms of its completion time, changes with the available network bandwidth. We encode the tasks in the job with different coding schemes, and compare their completion time with different available network bandwidth. In order to change the available network bandwidth, we use `iperf` to send additional traffic at a fixed throughput of 3 Gbps from another server to the master, which competes for the network bandwidth with all workers. With the additional traffic, the job will get less available bandwidth and need more time to finish. However, as we run the same job with different coding schemes, different coding schemes can be affected differently with the loss of available bandwidth, and we present two examples in Fig. 4.

In Fig. 4a, we first compare the performance of a $(2, 2)$ polynomial code with a $(2, 4, 2)$ entangled polynomial code. With the traffic described above, the entangled polynomial code completes the job 11.1% slower than the polynomial code. When such traffic is stopped, however, its time becomes 34.2% faster. We can also observe the same overtaking in Fig. 4c, between a MatDot code with $p = 2$ and a $(2, 1, 4)$ entangled polynomial code. The time of the MatDot code is also originally 6.7% faster when there is less available bandwidth, but becomes even 32.5% slower when there is no such traffic.

From the examples above, we can see that the two entangled polynomial codes can be more easily affected by the available bandwidth than the polynomial code and the MatDot code. This is because the master also needs to receive more data from workers before decoding, even if the task encoded by the entangled polynomial code has a lower complexity. As resource availability is subject to frequent changes, it is challenging to choose the optimal coding scheme and parameters in advance. Therefore, we propose a local re-encoding framework that will allow changing the coding scheme and/or the values of their parameters dynamically with marginal overhead.

As local re-encoding proposed in this paper allows changing the coding scheme and its parameters by re-encoding each task locally, we can easily skip the computation and communication required by re-encoding all tasks globally from scratch. As shown in Fig. 4b, if we re-encode all necessary tasks globally, the time spent will be even longer than the completion time with traffic. However, with local re-encoding, we can directly change the coding scheme with marginal overhead.

IV. Local Re-encoding for Entangled Polynomial Codes

In this section, we present a framework of local re-encoding for entangled polynomial codes.

Definition 1: The local re-encoding of a task originally encoded with an (m, n, p) entangled polynomial code converts it into a new one encoded with a $(\lambda_m m, \lambda_n n, \lambda_p p)$ entangled polynomial code without obtaining additional data, where λ_m, λ_n, and λ_p are positive integers.

More specifically, if a job is originally encoded with an (m, n, p) EP code, we are able to further split $\tilde{A}_{\text{EP}}(x)$ and $\tilde{B}_{\text{EP}}(x)$, and re-encode them directly into a new coded task which is equivalent to the task encoded with a $(\lambda_m m, \lambda_n n, \lambda_p p)$ EP code. First, we can see an illustrative example that re-encoding a task encoded with $p = 2$ MatDot code into a new task encoded with an $(m = 2, p = 2, n = 2)$ entangled polynomial code. The original partitions of matrix A and B with $p = 2$ MatDot code are $A = \begin{bmatrix} A_0 & A_1 \end{bmatrix}$ and $B = \begin{bmatrix} B_0 \\ B_1 \end{bmatrix}$, respectively. Therefore, the original coded tasks are $\tilde{A}_{\text{old}}(x) = A_0 x^0 + A_1 x^1$ and $\tilde{B}_{\text{old}}(x) = B_1 x^0 + B_0 x^1$. Through the global re-encoding, the master partitions the matrix A and B as $A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix}$ and $B = \begin{bmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{bmatrix}$ and encodes them with $(m = 2, p = 2, n = 2)$ entangled polynomial code directly from scratch to obtain the tasks, $\tilde{A}_{\text{EP}}(x) = A_{0,0} x^0 + A_{0,1} x^1 + A_{1,0} x^4 + A_{1,1} x^5$ and $\tilde{B}_{\text{EP}}(x) = B_{0,1} x^0 + B_{0,0} x^1 + B_{1,1} x^2 + B_{1,0} x^3$. Compared with re-encoding the original input matrices A and B in global re-encoding, local re-encoding further partitions the original coded tasks $\tilde{A}_{\text{old}}(x)$ and $\tilde{B}_{\text{old}}(x)$. In other words,

we have $\tilde{A}_{\text{old}}(x) = \begin{bmatrix} A_{0,0}x^0 + A_{0,1}x^1 \\ A_{1,0}x^0 + A_{1,1}x^1 \end{bmatrix}$ and $\tilde{B}_{\text{old}}(x) = \begin{bmatrix} B_{0,1}x^0 + B_{0,0}x^1 & B_{1,1}x^0 + B_{1,0}x^1 \end{bmatrix}$. Then each worker can run the encoding on the above partition to obtain the new coded tasks, $\tilde{A}_{\text{new}}(x) = (A_{0,0}x^0 + A_{0,1}x^1)x^0 + (A_{1,0}x^0 + A_{1,1}x^1)x^4$ and $\tilde{B}_{\text{new}}(x) = (B_{0,1}x^0 + B_{0,0}x^1)x^0 + (B_{1,1}x^0 + B_{1,0}x^1)x^2$. We can see that $\tilde{A}_{\text{EP}}(x) = \tilde{A}_{\text{new}}(x)$ and $\tilde{B}_{\text{EP}}(x) = \tilde{B}_{\text{new}}(x)$. In other words, local re-encoding can obtain the new coded tasks successfully. Moreover, while conventionally the original matrices A and B need to be re-encoded again from scratch, local re-encoding requires no additional data from any remote server, leading to marginal overhead. Saving the complexity of each task by $\lambda_m\lambda_n\lambda_p$ times, our framework achieves a flexible tradeoff between computation and communication overheads.

The change of coding schemes can also be supported by this result. From Sec. III-A we can easily verify that $\tilde{A}_{\text{P}}(x) = \tilde{A}_{\text{EP}}(x)$ when $p = 1$, and $\tilde{A}_{\text{MD}}(x) = \tilde{A}_{\text{EP}}(x)$ when $m = n = 1$. The same equivalence can also be found in $\tilde{B}$. Hence, as a special case, we can also change the coding schemes and the parameters of a polynomial code or MatDot code.

In the rest of this section, we will present this framework in detail. For convenience, we may omit EP in $\tilde{A}_{\text{EP}}(x)$ and $\tilde{B}_{\text{EP}}(x)$ in the rest of this section when there is no ambiguity, *i.e.*, $\tilde{A}(x) = \tilde{A}_{\text{EP}}(x)$ and $\tilde{B}(x) = \tilde{B}_{\text{EP}}(x)$.

A. Changing p to $\lambda_p p$

We first show that a task with an (m, n, p) EP code can be locally re-encoded into a task with an $(m, n, \lambda_p p)$ EP code. We assume that the two input matrices A and B are originally split into mp and np submatrices, *i.e.*, $A = [A_{i,j}]_{m\times p}$ and $B = [B_{i,j}]_{p\times n}$, and has been encoded into coded tasks with $\tilde{A}(x)$ and $\tilde{B}(x)$.

In order to re-encode $\tilde{A}(x)$ and $\tilde{B}(x)$, we will further split $\tilde{A}(x)$ vertically into λ_p submatrices, and $\tilde{B}(x)$ horizontally into λ_p submatrices. Hence, we define $\tilde{A}(x) = \begin{bmatrix} \tilde{A}_0(x) & \cdots & \tilde{A}_{\lambda_p-1}(x) \end{bmatrix}$ and $\tilde{B}(x) = \begin{bmatrix} \tilde{B}_0(x) \\ \vdots \\ \tilde{B}_{\lambda_p-1}(x) \end{bmatrix}$.

Since $\tilde{A}(x)$ and $\tilde{B}(x)$ are linear combinations of $A_{i,l}$ and $B_{l,j}$, $i \in [0, m-1]$, $j \in [0, n-1]$, and $k \in [0, p-1]$, we are also equivalently splitting them into p submatrices vertically and horizontally, respectively. In other words, we can re-write A and B as $A = A' = [A'_{i,j}]_{m\times(\lambda_p p)}$ and $B = B' = [B'_{i,j}]_{(\lambda_p p)\times n}$. In other words, we split $A_{i,k}$ as $\begin{bmatrix} A'_{i,\lambda_p k} & \cdots & A'_{i,\lambda_p k+\lambda_p-1} \end{bmatrix}$, and $B_{k,j}$ as $\begin{bmatrix} (B'_{\lambda_p k,j}) \\ \vdots \\ (B'_{\lambda_p k+\lambda_p-1,j}) \end{bmatrix}$. Then we have

$$\begin{aligned}
&\begin{bmatrix} \tilde{A}_0(x) & \cdots & \tilde{A}_{\lambda_p-1}(x) \end{bmatrix} = \tilde{A}(x) \\
&= \sum_{i=0}^{m-1}\sum_{k=0}^{p-1} A_{i,k}x^{pni+k} \\
&= \sum_{i=0}^{m-1}\sum_{k=0}^{p-1} \begin{bmatrix} A'_{i,\lambda_p k} & \cdots & A'_{i,\lambda_p k+\lambda_p-1} \end{bmatrix} \cdot x^{pni+k} \\
&= \begin{bmatrix} \sum_{i=0}^{m-1}\sum_{k=0}^{p-1} A'_{i,\lambda_p k}x^{pni+k} & \cdots \\ \sum_{i=0}^{m-1}\sum_{k=0}^{p-1} A'_{i,\lambda_p k+\lambda_p-1}x^{pni+k} \end{bmatrix},
\end{aligned} \tag{4}$$

and

$$\begin{aligned}
&\begin{bmatrix} \tilde{B}_0(x)^T & \cdots & \tilde{B}_{\lambda_p-1}(x)^T \end{bmatrix}^T = \tilde{B}(x) \\
&= \sum_{j=0}^{n-1}\sum_{k=0}^{p-1} B_{p-1-k,j}x^{pj+k} \\
&= \sum_{j=0}^{n-1}\sum_{k=0}^{p-1} \begin{bmatrix} (B'_{(p-1-k)\lambda_p,j})^T & \cdots \\ (B'_{(p-1-k)\lambda_p+\lambda_p-1,j})^T \end{bmatrix}^T x^{pj+k} \\
&= \begin{bmatrix} \sum_{j=0}^{n-1}\sum_{k=0}^{p-1}(B'_{(p-1-k)\lambda_p,j}x^{pj+k})^T & \cdots \\ \sum_{j=0}^{n-1}\sum_{k=0}^{p-1}(B'_{(p-1-k)\lambda_p+\lambda_p-1,j}x^{pj+k})^T \end{bmatrix}^T.
\end{aligned} \tag{5}$$

In other words, $\tilde{A}_l(x) = \sum_{i=0}^{m-1}\sum_{k=0}^{p-1} A'_{i,\lambda_p k+l}x^{pni+k}$ and $\tilde{B}_l(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{p-1} B'_{(p-1-k)\lambda_p+l,j}x^{pj+k}$, where $l = 0, \ldots, \lambda_p - 1$.

Now we can re-encode $\tilde{A}_i(x)$ and $\tilde{B}_i(x)$ into a new coded task. By defining $x = \sigma^{\lambda_p}$, we can rewrite $\tilde{A}_l(x)$ and $\tilde{B}_l(x)$ as $\tilde{A}_l(x) = \sum_{i=0}^{m-1}\sum_{k=0}^{p-1} A'_{i,\lambda_p k+l}\sigma^{(pni+k)\lambda_p}$, and $\tilde{B}_l(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{p-1} B'_{(p-1-k)\lambda_p+l,j}\sigma^{(pj+k)\lambda_p}$, $l = 0, \ldots, \lambda_p - 1$. Then we can re-encode $\tilde{A}(x)$ and $\tilde{B}(x)$ as

$$\begin{aligned}
&\sum_{l=0}^{\lambda_p-1} \tilde{A}_l(x)\sigma^l \\
&= \sum_{l=0}^{\lambda_p-1}\sum_{i=0}^{m-1}\sum_{k=0}^{p-1} A'_{i,\lambda_p k+l}\sigma^{(pni+k)\lambda_p+l} \\
&= \sum_{i=0}^{m-1}\sum_{k=0}^{\lambda_p p-1} A'_{i,k}\sigma^{\lambda_p pni+k},
\end{aligned} \tag{6}$$

and

$$\begin{aligned}&\sum_{l=0}^{\lambda_p-1}\tilde{B}_{\lambda_p-1-l}(x)\sigma^l\\=&\sum_{l=0}^{\lambda_p-1}\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}B'_{(p-1-k)\lambda_p+\lambda_p-1-l,j}\sigma^{(pj+k)\lambda_p+l}\\=&\sum_{j=0}^{n-1}\sum_{k=0}^{\lambda_pp-1}B'_{\lambda_pp-1-k,j}\sigma^{\lambda_ppj+k}.\end{aligned}\tag{7}$$

From (6) and (7) we can see that the task after re-encoding is equivalent to $\tilde{A}(x)$ and $\tilde{B}(x)$ encoded with an (m,n,λ_pp) EP code.

B. Changing m to $\lambda_m m$

Now we show that a task with an (m,n,p) EP code can be locally re-encoded into a task with a $(\lambda_m m,n,p)$ EP code. In this case, we will split $\tilde{A}(x)$ into λ_m submatrices horizontally, *i.e.*, $\tilde{A}(x)=\begin{bmatrix}\tilde{A}_0(x)\\ \vdots\\ \tilde{A}_{\lambda_m-1}(x)\end{bmatrix}$. Similar to the case above, A will also be equivalently split into $\lambda_m m$ submatrices, *i.e.*, $A=[A'_{i,j}]_{(\lambda_m m)\times p}$. In other words, we have $A_{x,z}=\begin{bmatrix}A'_{\lambda_m i,k}\\ \vdots\\ A'_{\lambda_m i+\lambda_m-1,k}\end{bmatrix}$, and then

$$\begin{aligned}&\begin{bmatrix}\tilde{A}_0(x)^T & \cdots & \tilde{A}_{\lambda_m-1}(x)^T\end{bmatrix}^T\\=&\tilde{A}(x)=\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}\\&\begin{bmatrix}(A'_{\lambda_m i,k})^T & \cdots & (A'_{\lambda_m i+\lambda_m-1,k})^T\end{bmatrix}^T\cdot x^{pni+k}\\=&\begin{bmatrix}\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}(A'_{\lambda_m i,k}x^{pni+k})^T\cdots\\ \sum_{i=0}^{m-1}\sum_{k=0}^{p-1}(A'_{\lambda_m i+\lambda_m-1,k}x^{pni+k})^T\end{bmatrix}^T.\end{aligned}\tag{8}\tag{9}$$

Since $\tilde{B}(x)$ is not a function of m, we only need to re-encode $\tilde{A}(x)$ when we adjust the value of m. When m is changed to $\lambda_m m$, we will re-encode $\tilde{A}(x)$ as

$$\begin{aligned}&\sum_{l=0}^{\lambda_m-1}\tilde{A}_l(x)x^{lpmn}\\=&\sum_{l=0}^{\lambda_m-1}\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}A'_{\lambda_m i+l,k}x^{pni+k+lpmn}\\=&\sum_{l=0}^{\lambda_m-1}\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}A'_{\lambda_m i+l,k}x^{pn(lm+i)+k}.\end{aligned}$$

Here, after re-encoding, we generate $\tilde{A''}(x)$ which is encoded by a $(\lambda_m m,n,p)$ EP code from a matrix A'' with rows in A switched as in (10).

$$\begin{aligned}A''=&\left[\begin{array}{ccc}A''_{0,0}&\cdots&A''_{0,p-1}\\A''_{1,0}&\cdots&A''_{1,p-1}\\\vdots&\vdots&\vdots\\A''_{m-1,0}&\cdots&A''_{m-1,p-1}\\\hline A''_{m,0}&\cdots&A''_{m,p-1}\\\vdots&\vdots&\vdots\\A''_{2m-1,0}&\cdots&A''_{2m-1,p-1}\\\hline\vdots&\vdots&\vdots\\\hline A''_{(\lambda_m-1)m,0}&\cdots&A''_{(\lambda_m-1)m,p-1}\\\vdots&\vdots&\vdots\\A''_{(\lambda_m-1)m+(m-1),0}&\cdots&A''_{(\lambda_m-1)m+(m-1),p-1}\end{array}\right]\\=&\left[\begin{array}{ccc}A'_{0,0}&\cdots&A'_{0,p-1}\\A'_{\lambda_m,0}&\cdots&A'_{\lambda_m,p-1}\\\vdots&\vdots&\vdots\\A'_{\lambda_m(m-1),0}&\cdots&A'_{\lambda_m(m-1),p-1}\\\hline A'_{1,0}&\cdots&A'_{1,p-1}\\\vdots&\vdots&\vdots\\A'_{\lambda_m(m-1)+1,0}&\cdots&A'_{\lambda_m(m-1)+1,p-1}\\\hline\vdots&\vdots&\vdots\\\hline A'_{\lambda_m-1,0}&\cdots&A'_{\lambda_m-1,p-1}\\\vdots&\vdots&\vdots\\A'_{\lambda_m(m-1)+\lambda_m-1,0}&\cdots&A'_{\lambda_m(m-1)+\lambda_m-1,p-1}\end{array}\right].\end{aligned}\tag{10}$$

Although the sequence of rows in A is switched, it will not change the result of multiplication after decoding as we can always switch the rows in the result back to the original order, *i.e.*,

$$\begin{aligned}&\sum_{l=0}^{\lambda_m-1}\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}A'_{\lambda_m i+l,k}x^{pn(lm+i)+k}\\=&\sum_{l=0}^{\lambda_m-1}\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}A''_{lm+i,k}x^{pn(lm+i)+k}\\=&\sum_{i=0}^{\lambda_m m-1}\sum_{k=0}^{p-1}A''_{i,k}x^{pni+k}.\end{aligned}$$

Therefore, $\left(\sum_{l=0}^{\lambda_m-1}\tilde{A}_l(x)x^{lpmn}\right)\cdot\left(\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}B_{p-1-k,j}x^{pj+k}\right)$ is a polynomial of degree $\lambda_m mnp+p-2$. Given $t\in[0,\lambda_m mnp+p-2]$ where t can be uniquely written as $t=pn(l_0m+i_0)+pj+s$, $l_0\in[0,\lambda_m-1]$, $i_0\in[0,m-1]$, $j\in[0,n-1]$, and $s\in[0,2p-2]$, the coefficient of x^t is $\sum_{l=0}^{p-1}A'_{\lambda_m i_0+l_0,l}B_{l,j}$ if $s=p-1$.

C. Changing n to $\lambda_n n$

Similar to Sec. IV-B, we assume that $\tilde{B}(x)$ will be further split into λ_n submatrices vertically, *i.e.*, $\tilde{B}(x) = \begin{bmatrix}\tilde{B}_0(x) & \cdots & \tilde{B}_{\lambda_n-1}(x)\end{bmatrix}$. It means that B is equivalently split into $\lambda_n n$ submatrices vertically, *i.e.*, $B = [B'_{i,j}]_{p\times(\lambda_n n)}$. In other words, $B_{k,j} = \begin{bmatrix}B'_{k,\lambda_n j} & \cdots & B'_{k,\lambda_n j+\lambda_n-1}\end{bmatrix}$. We can then have

$$\begin{aligned}
&\left[\tilde{B}_0(x)\cdots\tilde{B}_{\lambda_n-1}(x)\right] = \tilde{B}(x)\\
=&\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}\begin{bmatrix}B'_{p-1-k,\lambda_n j} & \cdots & B'_{p-1-k,\lambda_n j+\lambda_n-1}\end{bmatrix}x^{pj+k}\\
=&\Bigg[\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}B'_{p-1-k,\lambda_n j}x^{pj+k}\cdots\\
&\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}B'_{p-1-k,\lambda_n j+\lambda_n-1}x^{pj+k}\Bigg].
\end{aligned}$$

When we change n to $\lambda_n n$, we only need to re-encode $\tilde{B}(x)$ as $\sum_{l=0}^{\lambda_n-1}\tilde{B}_l(x)x^{lpmn}$. As we will show below, although it cannot be directly written as $\tilde{B}(x)$ with an $(m,\lambda_n n,p)$ EP code, we show that it is equivalent to an $(m,\lambda_n n,p)$ EP code, as they achieve the same recovery threshold. Since

$$\begin{aligned}
&\tilde{A}(x)\cdot\sum_{l=0}^{\lambda_n-1}\tilde{B}_l(x)x^{lpmn}\\
=&\tilde{A}(x)\cdot\left(\sum_{l=0}^{\lambda_n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}B_{p-1-k,\lambda_n j+l}x^{pj+k+lpmn}\right)\\
=&\left(\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}A_{i,k}x^{pni+k}\right)\cdot\\
&\left(\sum_{l=0}^{\lambda_n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}B_{p-1-k,\lambda_n j+l}x^{pj+k+lpmn}\right)\\
=&\sum_{x=0}^{m-1}\sum_{l=0}^{\lambda_n-1}\sum_{j=0}^{n-1}\sum_{s=0}^{2p-2}\sum_{k=\max\{0,s-p+1\}}^{\min\{p-1,s\}}\\
&A_{i,k}B_{p-1-k,\lambda_n j+l}x^{pmnl+pni+pj+s},
\end{aligned}\tag{11}$$

the degree of the polynomial above is $pmn(\lambda_n-1)+pn(m-1)+p(n-1)+2p-2 = pm\lambda_n n+p-2$, and it's the same as that of an $(m,\lambda_n n,p)$ EP code. However, after the interpolation, the sequence of submatrices in AB will be different from their order in the EP code and should be shuffled back.

In order to retrieve the original order of submatrices in AB, we first consider the order before re-encoding. From (3) we can see that for any $t\in[0,mnp+p-2]$, it can be uniquely written as $t = pni+pj+s$. When $s=p-1$, the coefficients of x^t are $\sum_{l=0}^{p-1}A_{i,l}B_{l,j}$. In (11), instead, we can uniquely rewrite the exponent of x^t as $t = pmnl+pni+pj+s$, $t\in[0,pmn\lambda_n n+p-2]$. When $s=p-1$, we can find $\sum_{k=0}^{p-1}A_{i,k}B_{p-1-k,\lambda_n j+l}$ as its coefficient.

D. Changing (m,n,p) to $(\lambda_m m,\lambda_n n,\lambda_p p)$

In general, when we need to change the values of m, n, and p at the same time, we can simply apply the three steps above individually. We note that when $\lambda_m\neq 1$ or $\lambda_n\neq 1$, we will not construct the exact $\tilde{A}(x)$ or $\tilde{B}(x)$. Rows in A are virtually shuffled when $\lambda_m\neq 1$. If $\lambda_n\neq 1$, $\tilde{B}(m,\lambda_n n,p)$ is not constructed exactly, but it can maintain the recovery threshold of the corresponding EP code.

Therefore, we will first change p to $\lambda_p p$, then m to $\lambda_m m$, and finally n to $\lambda_n n$. Assume that each task is originally encoded with an (m,n,p) EP code. If A and B are of sizes $\Lambda_m m\times\Lambda_p p$ and $\Lambda_p p\times\Lambda_n n$, then each task can be re-encoded into any $(\lambda_m m,\lambda_n n,\lambda_p p)$ EP code, if $\lambda_m|\Lambda_m$, $\lambda_n|\Lambda_n$, and $\lambda_p|\Lambda_p$. The more divisors Λ_m, Λ_n, and Λ_p have, the more EP codes can be re-encoded to.

Moreover, even though $\lambda_m/\lambda_n/\lambda_p$ is not a divisor of $\Lambda_m/\Lambda_n/\Lambda_p$, we can still add all-zero additional rows or columns into $\tilde{A}_i$ or/and $\tilde{B}_i$ so that they can be divisible. As $\tilde{A}_i$ and $\tilde{B}_i$ are linear combinations of submatrices in A and B, respectively, it is equivalent to adding additional rows or/and columns in A and B, which will only add additional rows or/and columns with zero elements but not change any existing element in the result. The overhead of such padding is at most $\left(1+\frac{\lambda_m}{\Lambda_m}\right)\left(1+\frac{\lambda_p}{\Lambda_p}\right)$ of $\tilde{A}_i$ and $\left(1+\frac{\lambda_n}{\Lambda_n}\right)\left(1+\frac{\lambda_p}{\Lambda_p}\right)$ of $\tilde{B}_i$, which is marginal if $\lambda_m\ll\Lambda_m$, $\lambda_n\ll\Lambda_n$, and $\lambda_p\ll\Lambda_p$. Since A and B are supposed to be large matrices, it is easy to satisfy such requirements.

E. Complexity Analysis

We now discuss the complexity of our framework, especially the complexity of re-encoding, and compare it with the complexity of the encoding and the complexity of the task. We find that the complexity of re-encoding is marginal compared with both of them.

Since the overhead of the addition is much cheaper than that of multiplication, we analyze the complexity as the number of multiplications. For convenience, we rewrite the sizes of A and B as $M\times P$ and $P\times N$, *i.e.*, $M=\Lambda_m m$, $N=\Lambda_n n$, and $P=\Lambda_p p$. Then the sizes of $\tilde{A}(x)$ and $\tilde{B}(x)$ are $\frac{M}{m}\times\frac{P}{p}$ and $\frac{P}{p}\times\frac{N}{n}$, respectively.

When we encode a task with an (m,n,p) EP code, both $\tilde{A}(x)$ and $\tilde{B}(x)$ are encoded as a linear combination of the mp submatrices in A and the np submatrices in B. Therefore, each element in A and B will be multiplied with a constant, and the complexity of $\tilde{A}(x)$ and $\tilde{B}(x)$ is $O(MP)$ and $O(NP)$, respectively. Moreover, the constants are powers of x, leading

to $pn(m-1)+(p-1)$ multiplications. However, this complexity can be ignored as we assume A and B are large matrices.

As a comparison, when we adjust the values of x, the complexity of re-encoding is much lower. When p changes to $\lambda_p p$, $\tilde{A}(x)$ and $\tilde{B}(x)$ should be further split into λ_p submatrices and re-encoded into their linear combinations. Hence, their numbers of multiplications are $O(\frac{MP}{mp})$ and $O(\frac{NP}{np})$, respectively. Similarly, when the value of m or n changes, its complexity is also $O(\frac{MP}{mp})$ or $O(\frac{NP}{np})$. Hence, in the general case when (m,n,p) is changed to $(\lambda_m m, \lambda_n n, \lambda_p p)$, the overall complexity is $O(\frac{MP}{mp}+\frac{NP}{np})$.

Given the sizes of $\tilde{A}(x)$ and $\tilde{B}(x)$, the complexity of the matrix multiplication in a task with an (m,n,p) EP code is $\frac{MNP}{mnp}$. After re-encoding, the complexity of the multiplication becomes $\frac{MNP}{\lambda_m m\cdot\lambda_n n\cdot\lambda_p p}$. Therefore, the complexity of a task, including re-encoding and the matrix multiplication, is $O\left(\frac{P}{p}\left(\frac{M}{m}+\frac{N}{n}+\frac{MN}{\lambda_m m\cdot\lambda_n n\cdot\lambda_p}\right)\right)=O(\frac{MNP}{\lambda_m m\cdot\lambda_n n\cdot\lambda_p p})$ if M and N are large. In other words, the complexity of re-encoding is also marginal to that of the task.

Compared with a job with a $(\lambda_m m, \lambda_n n, \lambda_p p)$ EP code, the decoding overhead of the EP code after re-encoding will be the same, since the new code will be equivalent to a $(\lambda_m m, \lambda_n n, \lambda_p p)$ EP code.

V. Local Re-encoding for Tang-Konstantinidis-Ramamoorthy Codes

In Sec. IV, we have proposed a framework of local re-encoding for EP codes, which are constructed based on polynomials, *i.e.*, the coded task is essentially a multiplication of two polynomials. Therefore, the decoding requires interpolating a polynomial from multiple evaluation points, as the results of matrix multiplication are located in the coefficients of this polynomial. Although it is numerically stable for polynomials on finite fields, as shown in the classical coding theory, it is not the case for matrix multiplication with real numbers. The interpolation of a polynomial effectively solves a linear system with a Vandermonde matrix, and Vandermonde matrices are well known to have large condition numbers [12], [30]. Therefore, small perturbations due to numerical precision errors can lead to large errors, especially for the polynomial with a large degree.

Compared with re-encoding all tasks from scratch, the numerical stability can be more easily affected after local re-encoding. If we re-encode all tasks from scratch, we can easily change the evaluation point of the polynomials in any task. However, the evaluation points chosen for the original polynomial cannot be changed after local re-encoding, and thus the evaluation points after local re-encoding may lead to arbitrarily high condition numbers in the corresponding Vandermonde matrix.

In this section, we present an extension of the local re-encoding framework, by supporting a variation of EP codes which maintains the numerical stability for bounded entries in the matrices and allows achieving a flexible tradeoff between the numerical stability and the computational overhead, without changing the recovery threshold. Hence, the error after re-encoding can be significantly lower than that with EP codes. Moreover, we can now change the complexity of a task without changing the recovery threshold, while maintaining the numerical stability.

A. Background and Examples

We now briefly introduce the code construction and its properties. For convenience, we name it as Tang-Konstantinidis-Ramamoorthy codes or TKR codes.[2] Interested readers may find more details of TKR codes in [13].

Assume that the input matrices A and B are originally split as $A=[A_{i,j}]_{m\times p}$ and $B=[B_{i,j}]_{p\times n}$. In addition, all entries in A and B are non-negative integers.[3] We first demonstrate a special case of TKR codes. Assuming that ζ is a large enough integer, A and B are then encoded as $\tilde{A}_{\text{TKR}}(x)=\sum_{i=0}^{m-1}\sum_{k=0}^{p-1}A_{i,k}\zeta^k x^{ni}$ and $\tilde{B}_{\text{TKR}}(x)=\sum_{j=0}^{n-1}\sum_{k=0}^{p-1}B_{k,j}\zeta^{-k}x^j$. In particular, the value of ζ is the same in all tasks, while the values of x should be different. Hence, we have $\tilde{A}_{\text{TKR}}(x)\tilde{B}_{\text{TKR}}(x)=\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left(\sum_{t=-(p-1)}^{p-1}\sum_{l=\max\{0,t\}}^{\min\{p-1,t+p-1\}}A_{i,l}B_{-t+l,j}\zeta^t\right)x^{ni+j}$.

Similar to polynomial codes, we can decode

$$\sum_{t=-(p-1)}^{p-1}\sum_{l=\max\{0,t\}}^{\min\{p-1,t+p-1\}}A_{i,l}B_{-t+l,j}\zeta^t, \tag{12}$$

where $i=0,\ldots,m-1, j=0,\ldots,n-1$, with mn tasks since the values of x are different. Furthermore, if ζ is large enough such that all entries in the coefficients are smaller than $\frac{\zeta}{2}$, Tang *et al.* have proved that $|\sum_{t=-(p-1)}^{-1}\sum_{l=\max\{0,t\}}^{\min\{p-1,t+p-1\}}A_{i,l}B_{-t+l,j}\zeta^t|<\frac{1}{2}$. Therefore, we can recover the coefficient of ζ^0, *i.e.*, $\sum_{l=0}^{p-1}A_{i,l}B_{l,j}$, by rounding (12) to the nearest integer and then computing the remainder upon division by ζ. Although the first step of decoding still involves a Vandermonde matrix, its degree is only $mn-1$, and the error can be further mitigated in the second step. Hence, it is shown in [13] that numerical errors of TKR codes are much smaller than those of EP codes.

We can see that the above example of TKR codes has a recovery threshold of mn, although the input matrices are

[2]Without modifying the code construction and its properties, we slightly change the notations in the code construction to conform with existing ones in this paper.

[3]Floating-point entries with limited precision can be handled with scaling.

split into mp and np submatrices. In fact, when $p = 1$, it downgrades to a polynomial code. TKR codes also support a more general recovery threshold by trading off numerical precision. Assume $p'|p$ and $q = \frac{p}{p'}$, we then have $\tilde{A}_{\text{TKR}}(x) = \sum_{i=0}^{m-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1} A_{i,qk+u}\zeta^u x^{p'ni+k}$ and $\tilde{B}_{\text{TKR}}(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1} B_{q(p'-1-k)+u,j}\zeta^{-u}x^{p'j+k}$. In other words, the special case above corresponds to $p' = 1$. Therefore,

$$\tilde{A}_{\text{TKR}}(x)\tilde{B}_{\text{TKR}}(x) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\sum_{t'=0}^{2p'-2}\sum_{l'=\max\{0,t'-p'+1\}}^{\min\{p'-1,t'\}} \left(\sum_{t=-(q-1)}^{q-1}\sum_{l=\max\{0,t\}}^{\min\{q-1,t+q-1\}} A_{i,ql'+l} B_{q(p'-1-t'+l')-t+l,j}\zeta^t\right) x^{p'ni+p'j+t'}. \tag{13}$$

As a polynomial of x, (13) has a degree of $mnp' + p' - 2$ and hence has a recovery threshold of $mnp' + p' - 1$. After interpolation, we can get $\sum_{l'=\max\{0,t'-p'+1\}}^{\min\{p'-1,t'\}}\sum_{t=-(q-1)}^{q-1}\sum_{l=\max\{0,t\}}^{\min\{q-1,t+q-1\}} A_{i,ql'+l}\cdot B_{q(p'-1-t'+l')-t+l,j}\zeta^t$ as the coefficient of $x^{p'ni+p'j+t'}$. We then round it to the nearest integer and compute the remainder upon division by ζ. Eventually we will obtain $\sum_{l'=0}^{p'-1}\sum_{l=0}^{q-1} A_{i,ql'+l}B_{ql'+l,j} = \sum_{k=0}^{p-1} A_{i,k}B_{k,j}$, $i = 0,\ldots,m-1$, $j = 0,\ldots,n-1$.

In this paper, we show that with a non-trivial extension, our local re-encoding framework can also be applied to TKR codes.

Definition 2: The local re-encoding of a task originally encoded with an (m, n, p, p') TKR code converts it into a new one encoded with a $(\lambda_m m, \lambda_n n, \lambda_p p, \lambda_{p'} p')$ TKR code without obtaining additional data, where $\lambda_m, \lambda_n, \lambda_p$, and $\lambda_{p'}$ are positive integers, and $\lambda_{p'}|\lambda_p$.

Note that when $p = p'$, the corresponding TKR code downgrades to an (m, n, p) entangled polynomial code. As a special case of this result, we can locally re-encode tasks from the entangled polynomial code (or its special cases including the polynomial code and MatDot code) to the TKR code. Similarly, we first show an illustrative example that re-encoding a task encoded with $(m = 1, p = 2, n = 1, p' = 2)$ TKR code into a new task encoded with $(m = 1, p = 4, n = 1, p' = 2)$ TKR code. In the original task, the partitions of A and B are $A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$ and $B = \begin{bmatrix} B_0 & B_1 \end{bmatrix}$, so we have the coded tasks $\tilde{A}_{\text{old}}(x) = A_0\zeta^0 x^0 + A_1\zeta^0 x^1$ and $\tilde{B}_{\text{old}}(x) = B_0\zeta^0 x^0 + B_1\zeta^0 x^1$. By global re-encoding, the master partitions the input matrices A and B as $A = \begin{bmatrix} A_0^T & A_1^T & A_2^T & A_3^T \end{bmatrix}^T$ and $B = \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \end{bmatrix}$, respectively. Therefore, the master can re-encode A and B from scratch to obtain the new coded tasks $\tilde{A}_{\text{TKR}}(x) = A_0\zeta^0 x^0 + A_1\zeta^1 x^0 + A_2\zeta^0 x^1 + A_3\zeta^1 x^1$ and $\tilde{B}_{\text{TKR}}(x) = B_0\zeta^0 x^1 + B_1\zeta^{-1}x^1 + B_2\zeta^0 x^0 + B_3\zeta^{-1}x^0$. With local re-encoding framework, each worker locally further partition the original coded tasks $\tilde{A}_{\text{old}}(x)$ vertically and $\tilde{B}_{\text{old}}(x)$ horizontally. Using the local re-encoding formula presented in Sec. V-B, we have the new coded tasks $\tilde{A}_{\text{new}}(x) = A_0\zeta^0 x^0 + A_1\zeta^1 x^0 + A_2\zeta^0 x^1 + A_3\zeta^1 x^1$ and $\tilde{B}_{\text{new}}(x) = B_0\zeta^0 x^1 + B_1\zeta^{-1}x^1 + B_2\zeta^0 x^0 + B_3\zeta^{-1}x^0$. We can see that $\tilde{A}_{\text{TKR}}(x) = \tilde{A}_{\text{new}}(x)$ and $\tilde{B}_{\text{TKR}}(x) = \tilde{B}_{\text{new}}(x)$. Therefore, the proposed local re-encoding framework can obtain the same coded tasks as those encoded with the general TKR code globally.

We can also see that in TKR codes the recovery threshold does not change when p increases. In other words, with the same values of m, n, and p', TKR codes achieve a tradeoff between the numerical stability and the complexity of the task. Tang *et al.* have reported that numerical precision decreases, *i.e.*, the error after decoding increases when the value of p increases [13]. Hence, we can also demonstrate that our framework achieves a flexible tradeoff between numerical precision and the complexity of the task (and hence job completion time) through local re-encoding.

For convenience, we may also omit TKR in this section when there is no ambiguity, *i.e.*, $\tilde{A}(x) = \tilde{A}_{\text{TKR}}(x)$ and $\tilde{B}(x) = \tilde{B}_{\text{TKR}}(x)$.

B. Changing (m, n, p, p') to $(m, n, \lambda_p p, p')$

As described above, the recovery threshold of TKR codes, *i.e.*, $mnp' + p' - 1$ does not depend on p. Hence, different from the local re-encoding for EP codes in Sec. IV where the recovery threshold must be changed after re-encoding, it is possible to re-encode a task encoded from TKR codes without changing the recovery threshold. In other words, we can flexibly achieve a different tradeoff between numerical precision and task complexity.

In order to achieve a different tradeoff, we need to further split $\tilde{A}(x)$ vertically and $\tilde{B}(x)$ horizontally, *i.e.*, $\tilde{A}(x) = \begin{bmatrix} \tilde{A}_0(x) & \cdots & \tilde{A}_{\lambda_p-1}(x) \end{bmatrix}$ and $\tilde{B}(x) = \begin{bmatrix} \tilde{B}_0(x) \\ \vdots \\ \tilde{B}_{\lambda_p-1}(x) \end{bmatrix}$. Equivalently, by splitting $\tilde{A}(x)$ and $\tilde{B}(x)$, we are also splitting A and B as Sec. IV-A. Similar to (4) and (5), we can get $\tilde{A}_l(x) = \sum_{i=0}^{m-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1} A'_{i,\lambda_p(qk+u)+l}\zeta^u x^{p'ni+k}$ and $\tilde{B}_l(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1} B'_{\lambda_p(q(p'-1-k)+u)+l,j}\zeta^{-u}x^{p'j+k}, l = 0,...,\lambda_p-1$.

We can now re-encode $\tilde{A}(x)$ and $\tilde{B}(x)$ as:

$$\sum_{l=0}^{\lambda_p-1}\tilde{A}_l(x)\zeta^{ql}=\sum_{l=0}^{\lambda_p-1}\sum_{i=0}^{m-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1}A'_{i,\lambda_p(qk+u)+l}\cdot \zeta^{u+ql}x^{p'ni+k},$$

and

$$\sum_{l=0}^{\lambda_p-1}\tilde{B}_l(x)\zeta^{-ql}=\sum_{l=0}^{\lambda_p-1}\sum_{j=0}^{n-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1}B'_{\lambda_p(q(p'-1-k)+u)+l,j}\cdot \zeta^{-u-ql}x^{p'j+k}.$$

Although the task after re-encoding is not exactly the same as an $(m,n,\lambda_p p,p')$ TKR code, we can show that after multiplication, we can still decode the result of AB with the same recovery threshold, *i.e.*, it is equivalent to an $(m,n,\lambda_p p,p')$ TKR code.

Given A partitioned into $m\times\lambda_p p$ submatrices and B partitioned into $\lambda_p p\times n$ submatrices, *i.e.*, $A=[A_{i,j}]_{m\times(\lambda_p p)}$ and $B=[B_{i,j}]_{(\lambda_p p)\times\lambda_n}$, we shuffle the columns of A and rows of B such that the result of matrix multiplication remains unchanged. Given any $k\in[0,\lambda_p p-1]$, it can be uniquely written as $v=q(\lambda_p k+l)+u$ where $l\in[0,\lambda_p-1]$, $u\in[0,q-1]$, and $k\in[0,p'-1]$. We define a mapping function $f(v)=\lambda_p(qk+u)+l$ and map A to $A''=[A''_{i,j}]_{m\times(\lambda_p p)}=[A'_{i,f(j)}]_{m\times(\lambda_p p)}$. We also define $B''=[B''_{i,j}]_{(\lambda_p p)\times n}=[B'_{f(i),j}]_{(\lambda_p p)\times n}$. Therefore, $AB=A'B'=A''B''$ as we shuffle columns of A' and rows of B' in the same way.

Applying an $(m,n,\lambda_p p,p')$ TKR code on A'', we have $\tilde{A''}(x)=\sum_{i=0}^{m-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{\lambda_p q-1}A''_{i,\lambda_p qk+u}\zeta^u x^{p'ni+k}$. In particular, as we can uniquely have k_0 and l such that $u=ql+u_0$ where $l\in[0,\lambda_p-1]$ and $u_0\in[0,q-1]$, we then have $A''_{i,\lambda_p qk+u}=A'_{i,f(\lambda_p qk+ql+u_0)}=A'_{i,\lambda_p(qk+u_0)+l}$. Hence, $\tilde{A''}(x)=\sum_{i=0}^{m-1}\sum_{k_0=0}^{p'-1}\sum_{l=0}^{\lambda_p-1}\sum_{u_0=0}^{q-1}A'_{i,\lambda_p(qk_0+u_0)+l}\cdot\zeta^{ql+u_0}x^{p'ni+k}$.

Similarly, we have

$$\begin{aligned}\tilde{B''}(x)&=\sum_{j=0}^{n-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{\lambda_p q-1}B''_{\lambda_p q(p'-1-k)+u,j}\zeta^{-u}x^{p'j+k}\\&=\sum_{j=0}^{n-1}\sum_{k=0}^{p'-1}\sum_{l=0}^{\lambda_p-1}\sum_{u_0=0}^{q-1}B'_{f(\lambda_p q(p'-1-k)+ql+u_0),j}\cdot\\&\quad\zeta^{-(ql+u_0)}x^{p'j+k}\\&=\sum_{j=0}^{n-1}\sum_{k_0=0}^{p'-1}\sum_{l=0}^{\lambda_p-1}\sum_{u_0=0}^{q-1}B'_{\lambda_p(q(p'-1-k_0)+u_0)+l,j}\cdot\\&\quad\zeta^{-(ql+u_0)}x^{p'j+k}.\end{aligned}$$

The equations above show that the re-encoded task is equivalent to that encoded with an $(m,n,\lambda_p p,p')$ TKR code. We can also see that after re-encoding, the recovery threshold remains unchanged as $mnp'+p'-1$. However, the sizes of $\tilde{A}$ and $\tilde{B}$ are reduced by λ_p times, reducing the complexity of the multiplication by λ_p times. As we will demonstrate in Sec. VI-C, the tradeoff between the complexity of the task and the numerical precision can be achieved without changing the number of workers.

C. Changing (m,n,p,p') to $(\lambda_m m,\lambda_n n,\lambda_p p,\lambda_{p'}p')$

By extending the special case above, we now demonstrate the general case of local re-encoding for TKR codes, achieving the most flexible tradeoff among the recovery threshold, task complexity, and numerical precision. In particular, when $p=p'$ and $\lambda_p=\lambda_{p'}$, it downgrades to local re-encoding of EP codes from (m,n,p) to $(\lambda_m,\lambda_n,\lambda_p)$. Instead of completing local re-encoding in three steps as in Sec. IV-D, we can now complete it in one step directly.

Before re-encoding, we first split $\tilde{A}(x)$ and $\tilde{B}(x)$ into $\lambda_m\lambda_p$ and $\lambda_n\lambda_p$ submatrices, *i.e.*, $\tilde{A}(x)=[\tilde{A}_{i,j}(x)]_{\lambda_m\times\lambda_p}$ and $\tilde{B}(x)=[\tilde{B}_{i,j}(x)]_{\lambda_p\times\lambda_n}$. Correspondingly, A and B should also be further partitioned into $\lambda_m m\cdot\lambda_p p$ and $\lambda_n n\cdot\lambda_p p$ submatrices, *i.e.*, $A_{i,k}=[A'_{\lambda_m i+t_0,\lambda_p k+t_1}]_{\lambda_m\times\lambda_p}$ and $B_{k,j}=[B'_{\lambda_p k+t_0,\lambda_n j+t_1}]_{\lambda_p\times\lambda_n}$. In addition, we have $\tilde{A}_{t_0,l}(x)=\sum_{i=0}^{m-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1}A'_{\lambda_m i+t_0,\lambda_p(qk+u)+l}\zeta^u x^{p'ni+k}$ and $\tilde{B}_{l,t_1}(x)=\sum_{j=0}^{n-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1}B'_{\lambda_p(q(p'-1-k)+u)+l,\lambda_n j+t_1}\cdot\zeta^{-u}x^{p'j+k}$, where $t_0=0,\dots,\lambda_m-1$, $t_1=0,\dots,\lambda_n-1$, and $l=0,\dots,\lambda_p-1$.

Given $v\in[0,\lambda_p p-1]$, there is a uniquely tuple (z_0,u,l_1,l_0) where $z_0\in[0,p'-1]$, $u\in[0,q-1]$, $l_1\in[0,\lambda_{p'}-1]$, and $l_0\in[0,\lambda_p-1]$, such that $v=\lambda_p qz_0+\lambda_q ql_0+ql_1+u$, and we define $f(v)=\lambda_p qz_0+\lambda_p u+\lambda_{p'}l_1+l_0$. Using $f(v)$, we can map A' and B' into A'' and B'' such that $A''=[A''_{i,j}]_{\lambda_m m\times(\lambda_p p)}=[A'_{i,f(j)}]_{\lambda_m m\times(\lambda_p p)}$ and $B''=[B''_{i,j}]_{(\lambda_p p)\times(\lambda_n n)}=[B'_{f(i),j}]_{(\lambda_p p)\times\lambda_n n}$, and still have $AB=A'B'=A''B''$.

We assume $x=x_{\text{new}}^{\lambda_{p'}}$, and define $\lambda_q=\frac{\lambda_p}{\lambda_{p'}}$. We also shuffle the columns of A' and rows of B' to construct A'' and B''. Hence, $\tilde{A}(x)$ and $\tilde{B}(x)$ can be re-encoded as

$$\sum_{l=0}^{\lambda_m-1}x_{\text{new}}^{l\lambda_{p'}p'mn}\left(\sum_{l_0=0}^{\lambda_{p'}-1}\sum_{l_1=0}^{\lambda_q-1}\tilde{A}_{l,\lambda_{p'}l_1+l_0}(x)\zeta^{ql_1}x_{\text{new}}^{l_0}\right)\tag{14}$$

$$=\sum_{l=0}^{\lambda_m-1}x_{\text{new}}^{l\lambda_{p'}p'mn}\left(\sum_{l_0=0}^{\lambda_{p'}-1}\sum_{l_1=0}^{\lambda_q-1}\sum_{i=0}^{m-1}\sum_{k=0}^{p'-1}\sum_{u=0}^{q-1}A'_{\lambda_m i+l,\lambda_p(qk+u)+\lambda_{p'}l_1+l_0}\zeta^u x^{p'ni+k}\zeta^{ql_1}x_{\text{new}}^{l_0}\right)$$

$$
\begin{aligned}
&= \sum_{l=0}^{\lambda_m-1} x_{\text{new}}^{l\lambda_{p'}p'mn} \Bigg(\sum_{i=0}^{m-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} \\
&\quad A''_{\lambda_m i+l,\lambda_q qk+u} \zeta^u x_{\text{new}}^{\lambda_{p'}p'ni+k} \Bigg) \\
&= \sum_{l=0}^{\lambda_m-1} \sum_{i=0}^{m-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} A''_{\lambda_m i+l,\lambda_q qk+u} \cdot \\
&\quad \zeta^u x_{\text{new}}^{\lambda_{p'}p'n(lm+i)+k} \\
&= \sum_{l=0}^{\lambda_m-1} \sum_{i=0}^{m-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} A'''_{i+lm,\lambda_q qk+u} \cdot \\
&\quad \zeta^u x_{\text{new}}^{\lambda_{p'}p'n(lm+i)+k} \\
&= \sum_{i=0}^{\lambda_m m-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} A'''_{i,\lambda_q qk+u} \zeta^u x_{\text{new}}^{\lambda_{p'}p'ni+k},
\end{aligned} \tag{15}
$$

where A''' is constructed by shuffling the rows of A'' as in (10), and

$$
\begin{aligned}
&\sum_{l=0}^{\lambda_n-1} x_{\text{new}}^{l\lambda_{p'}p'\lambda_m mn} \Bigg(\sum_{l_0=0}^{\lambda_{p'}-1} \sum_{l_1=0}^{\lambda_q-1} \tilde{B}_{\lambda_{p'}l_1+l_0,l}(x) \cdot \\
&\quad \zeta^{-ql_1} x_{\text{new}}^{\lambda_{p'}-1-l_0} \Bigg) \\
&= \sum_{l=0}^{\lambda_n-1} x_{\text{new}}^{l\lambda_{p'}p'\lambda_m mn} \Bigg(\sum_{l_0=0}^{\lambda_{p'}-1} \sum_{l_1=0}^{\lambda_q-1} \sum_{j=0}^{n-1} \sum_{k=0}^{p'-1} \sum_{u=0}^{q-1} \\
&\quad B'_{\lambda_p(q(p'-1-k)+u)+\lambda+p'l_1+l_0,\lambda_n j+l} \zeta^{-u} x^{p'j+k} \cdot \\
&\quad \zeta^{-ql_1} x_{\text{new}}^{\lambda_{p'}-1-l_0} \Bigg) \\
&= \sum_{l=0}^{\lambda_n-1} x_{\text{new}}^{l\lambda_{p'}p'\lambda_m mn} \Bigg(\sum_{j=0}^{n-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} \\
&\quad B''_{\lambda_q q(\lambda_{p'}p'-1-k)+u,\lambda_n j+l} \zeta^{-u} x_{\text{new}}^{\lambda_{p'}p'j+k} \Bigg) \\
&= \sum_{l=0}^{\lambda_n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} B''_{\lambda_q q(\lambda_{p'}p'-1-k)+u,\lambda_n j+l} \cdot \\
&\quad \zeta^{-u} x_{\text{new}}^{\lambda_{p'}p'(l\lambda_m mn+j)+k}.
\end{aligned} \tag{16}
$$

Therefore, we have

$$
\begin{aligned}
&\Bigg(\sum_{i=0}^{\lambda_m m-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} A'''_{i,\lambda_q qk+u} \zeta^u x_{\text{new}}^{\lambda_{p'}p'ni+k} \Bigg) \cdot \\
&\Bigg(\sum_{l_1=0}^{\lambda_n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{\lambda_p p'-1} \sum_{u=0}^{\lambda_q q-1} B''_{\lambda_q q(\lambda_{p'}p'-1-k)+u,\lambda_n j+l_1} \cdot \\
&\quad \zeta^{-u} x_{\text{new}}^{\lambda_{p'}p'(l_1\lambda_m mn+j)+k} \Bigg) \\
&= \sum_{i=0}^{\lambda_m m-1} \sum_{l_1=0}^{\lambda_n-1} \sum_{j=0}^{n-1} \sum_{t'=0}^{2(\lambda_{p'}p'-1)} \sum_{l'=\max\{0,t'-\lambda_{p'}p'+1\}}^{\min\{\lambda_{p'}p'-1,t'\}} \\
&\quad \Bigg(\sum_{t=-(\lambda_q q-1)}^{\lambda_q q-1} \sum_{l=\max\{0,t\}}^{\min\{\lambda_q q-1,t+\lambda_q q-1\}} A'''_{i,\lambda_q ql'+l} \cdot \\
&\quad B''_{\lambda_q q(\lambda_{p'}p'-1-t'+l')-t+l,\lambda_n j+l_1} \zeta^t \Bigg) \cdot \\
&\quad x_{\text{new}}^{\lambda_{p'}p'ni+\lambda_{p'}(p'j+l_1 p'\lambda_m mn)+t'}.
\end{aligned}
$$

If we consider the equation above as a polynomial of x_{new}, its degree is $\lambda_m m\lambda_n n\lambda_{p'}p' + \lambda_{p'}p' - 2$. Hence with any $\lambda_m m\lambda_n n\lambda_{p'}p' + \lambda_{p'}p' - 1$ tasks, we can obtain all its coefficients. In particular, given a $i \in [0, \lambda_m m - 1]$, $j \in [0, \lambda_n - 1]$, $s \in [0, n-1]$, the coefficients when $t' = \lambda_{p'}p' - 1$ are $\sum_{l'=0}^{\lambda_{p'}p'-1} \sum_{t=-(\lambda_q q-1)}^{\lambda_q q-1} \sum_{l=\max\{0,t\}}^{\min\{\lambda_q q-1,t+\lambda_q q-1\}} A'''_{i,\lambda_q ql'+l} \cdot B''_{\lambda_q q(\lambda_{p'}p'-1-t'+l')-t+l,\lambda_n s+j} \zeta_{\text{new}}^t$, which can be considered as a polynomial of ζ_{new}. When ζ_{new} is large enough, we can obtain the coefficient of ζ_{new}^0, *i.e.*, $\sum_{l'=0}^{\lambda_{p'}p'-1} \sum_{l=0}^{\lambda_q q-1} A'''_{i,\lambda_q ql'+l} B''_{\lambda_q ql'+l,\lambda_n s+j} = \sum_{k=0}^{\lambda_p p-1} A'''_{i,k} \cdot B''_{k,s'}$ where $k = \lambda_q ql' + l$ and $s' = \lambda_n s + j \in [0, \lambda_n n - 1]$. Hence, we can obtain the result of $A'''B''$. Since A''' is constructed by shuffling the rows of A'', we also obtain the result of $A''B'' = AB$.

D. Complexity Analysis

The same as EP codes, the re-encoding framework for TKR codes also maintains the same decoding complexity as the new task is proved to be equivalent as one encoded with a $(\lambda_m m, \lambda_n n, \lambda_p p, \lambda_{p'}p')$ TKR code.

As for the complexity of re-encoding, we can see from (14) and (16) that it is also a linear combination of the $\lambda_m\lambda_p$ and $\lambda_p\lambda_n$ submatrices in $\tilde{A}(x)$ and $\tilde{B}(x)$. Hence, the complexity of local re-encoding is still $O(\frac{MP}{mp} + \frac{NP}{np})$, which is still marginal to the complexity of the task.

VI. Evaluation

To demonstrate the performance of local re-encoding, we still implement our framework with Open MPI [29]. If local re-encoding is needed before a worker executes its task, it will first re-encode $\tilde{A}(x)$ and $\tilde{B}(x)$ locally, then multiply the two re-encoded matrices, and finally upload the result to the master. As a comparison, we also allow the tasks to be re-encoded *globally*, *i.e.*, all tasks will be encoded again from scratch from A and B. The new tasks will then be sent to all workers, and all workers will then start to run such new tasks.

A. Overhead of Re-encoding

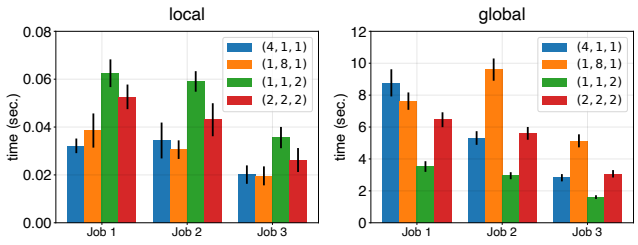
We compare the overhead of re-encoding between local re-encoding and global re-encoding for EP codes and TKR codes

in terms of the time consumption. As for global re-encoding, to change the coding scheme, we need to encode all tasks again from scratch, deploy such tasks to workers, and then start the job. Hence, the time of global re-encoding should include such three steps. The proposed local re-encoding, on the other hand, allows tasks to be re-encoded with a new coding scheme on local workers directly, and hence its time is measured as the time that all workers complete their local re-encoding.

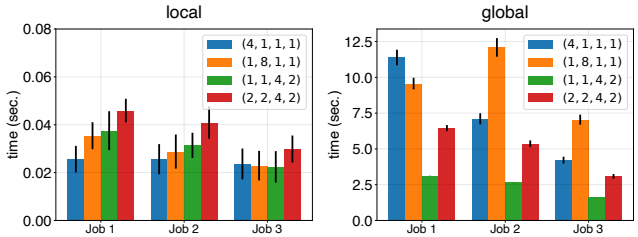
TABLE I: Sizes of input matrices in three jobs of matrix multiplications.

	Job 1	Job 2	Job 3
A	1024×2048	2048×2048	2048×1024
B	2048×4096	2048×2048	1024×2048

We run our experiments in Microsoft Azure. The master runs on a virtual machine of type `B4ms`, and all workers run on virtual machines of type `B1ms`. We set initial parameters of the EP code as $(2, 2, 2)$ and encode input matrices of three jobs. We run three jobs in our experiments, and the sizes of input matrices of such three jobs are shown in Table I, except that all entries are integers. Similarly, we also encode the three jobs with a $(2, 2, 2, 2)$ TKR code so that they are split in the same way as the $(2, 2, 2)$ EP code, and thus they have the same recovery threshold. In each job with EP codes, we change the parameters with four configurations: $(\lambda_m, \lambda_n, \lambda_p) = (4, 1, 1), (1, 8, 1), (1, 1, 2)$, and $(2, 2, 2)$. As for TKR codes, we change the parameters similarly: $(\lambda_m, \lambda_n, \lambda_p, \lambda_{p'}) = (4, 1, 1, 1), (1, 8, 1, 1), (1, 1, 4, 2)$, and $(2, 2, 4, 2)$. Hence, the recovery thresholds of TKR codes after local re-encoding remain the same as those of EP codes.



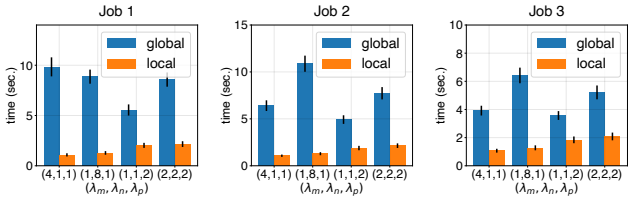
(a) EP codes



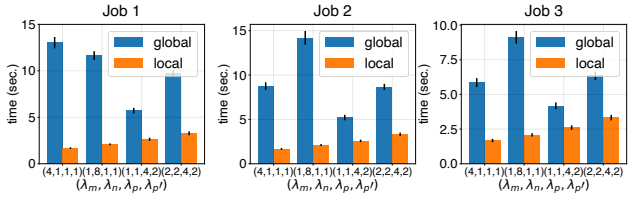
(b) TKR codes

Fig. 5: Overhead of re-encoding of EP codes and TKR codes.

With each configuration, we repeat every job 50 times and obtain the mean and standard deviation of its results. As for local re-encoding, the overhead of re-encoding comes only from re-encoding $\tilde{A}(x)$ and $\tilde{B}(x)$ locally. The overhead of global re-encoding, however, comprises the overhead of encoding which is performed solely at the master, and the overhead of distributing all new coded tasks. Therefore, although originally the time of global re-encoding with EP codes ranges between 1.90 seconds and 17.95 seconds, local re-encoding only needs 0.22 seconds on average at most, as shown in Fig. 5a. Similarly, Fig. 5b illustrates the overhead of re-encoding with TKR codes. We can see that, for local re-encoding, the maximum value in Fig. 5b is 0.05 seconds, which occurred in Job 1 with the configuration $(2, 2, 4, 2)$. As for global re-encoding, however, the time of re-encoding ranges between 1.61 seconds and 12.09 seconds. In general, Fig. 5 shows that the overhead of re-encoding can be saved by at most 99.37% for EP codes and 99.76% for TKR codes by local re-encoding.



(a) EP codes



(b) TKR codes

Fig. 6: Job completion time with local and global re-encoding.

We also evaluate how re-encoding affects the overall job completion time in Fig. 6. Specifically, job completion time includes the time of global or local re-encoding, computation of coded tasks on workers, and decoding on the master. Compared with job completion time in Fig. 6, we can see that the re-encoding overhead of local in Fig. 5 is marginal. Due to the saved re-encoding overhead, we can also see that local re-encoding can save job completion time by up to 88.59% with EP codes in Fig. 6a, and up to 87.05% with TKR codes in Fig. 6b.

B. *Numerical Stability*

We now compare the numerical stability with EP codes and TKR codes before and after re-encoding. For a fair comparison,

the entries of all matrices are integers. First we run the three jobs in Table I with a $(2, 2, 2)$ EP code, and then re-encode such jobs with $\lambda_m = 2$, $\lambda_n = 2$, $\lambda_p = 2$, and $\lambda_m = \lambda_n = \lambda_p = 2$. All other parameters, if they are not mentioned, remain unchanged. To evaluate the numerical stability, we compare the error of each job before re-encoding and after re-encoding. The number of workers is still chosen to tolerate at most 5 stragglers after re-encoding. The error is measured as the Frobenius norm, *i.e.*, $e = \frac{\|C-\hat{C}\|_F}{\|A\|_F * \|B\|_F}$, where C is the precise result of AB, and $\hat{C}$ is the result obtained after decoding [31].

The results of the errors are shown in Table II, where the data of errors were obtained as the averages of running the three jobs 50 times on Microsoft Azure. We can see from Table II that before re-encoding the three jobs all have very low numerical errors. After re-encoding, the errors are significantly increased. Compared with global re-encoding, the errors of local re-encoding are even higher, as the choices of xs cannot be changed with local re-encoding, making them less desirable for the new $(\lambda_m m, \lambda_n n, \lambda_p p)$ EP codes.

TABLE II: Comparison of errors of EP and TKR codes before and after re-encoding.

EP		Job 1	Job 2	Job 3
before re-encoding		1.78×10^{-9}	9.54×10^{-10}	1.31×10^{-9}
global	$\lambda_m = 2$	5.28×10^{-3}	5.90×10^{-3}	6.79×10^{-3}
	$\lambda_n = 2$	4.99×10^{-3}	7.67×10^{-3}	5.77×10^{-3}
	$\lambda_p = 2$	5.62×10^{-3}	5.82×10^{-3}	6.41×10^{-3}
	$\lambda_m = \lambda_n = \lambda_p = 2$	2.51×10^{-2}	2.61×10^{-2}	3.10×10^{-2}
local	$\lambda_m = 2$	1.87×10^{-2}	1.70×10^{-2}	2.21×10^{-2}
	$\lambda_n = 2$	1.61×10^{-2}	1.65×10^{-2}	2.18×10^{-2}
	$\lambda_p = 2$	1.56×10^{-2}	1.60×10^{-2}	1.14×10^{-2}
	$\lambda_m = \lambda_n = \lambda_p = 2$	3.53×10^{-2}	3.36×10^{-2}	3.66×10^{-2}
TKR		Job 1	Job 2	Job 3
before re-encoding		0	0	0
global	$\lambda_m = 2$	0	0	0
	$\lambda_n = 2$	0	0	0
	$\lambda_p = 2$	0	0	0
	$\lambda_m = \lambda_n = \lambda_p = 2$	0	0	0
local	$\lambda_m = 2$	0	0	0
	$\lambda_n = 2$	0	0	0
	$\lambda_p = 2$	0	0	0
	$\lambda_m = \lambda_n = \lambda_p = 2$	0	0	0

We also demonstrate that the numerical stability of the three jobs with TKR codes before and after re-encoding in Table II. The three jobs are now originally encoded with a $(2, 2, 2, 2)$ TKR code so that they are split in the same way as the $(2, 2, 2)$ EP code above. In each job, we change the parameters with four configurations: $\lambda_m = 2$, $\lambda_n = 2$, $\lambda_p = 2$, and $\lambda_m = \lambda_n = \lambda_p = 2$. Since changing p' does not change the way how matrices are split, we always have $\lambda_{p'} = 1$ in this experiment. We will discuss how changing $\lambda_{p'}$ can lead to a tradeoff between completion time and numerical stability in Sec. VI-C. We can see that the numerical stability of TKR codes after local re-encoding can be maintained — the errors remain as 0 for different values of parameters. Compared with the errors of EP codes, we can see that the numerical stability is significantly improved by TKR codes, and meanwhile local re-encoding does not hurt the numerical stability, as opposed to EP codes.

C. The Tradeoff between Completion Time and Numerical Stability

Besides achieving high numerical stability, TKR codes allow changing the complexity of the task without changing the recovery threshold. From Sec. V-B, we can see that a task with an (m, n, p, p') TKR code can be re-encoded into a task with an $(m, n, \lambda_p p, p')$ TKR code. Hence, the sizes of the two coded matrices in the task are reduced by λ_p times, reducing the complexity of the task by λ_p times as well. Since the recovery threshold of the TKR code does not depend on p, the recovery threshold does not change after re-encoding. However, Tang *et al.* have pointed out that the error will increase when p is increased [13]. In other words, there is a tradeoff between completion time and the error of the job.

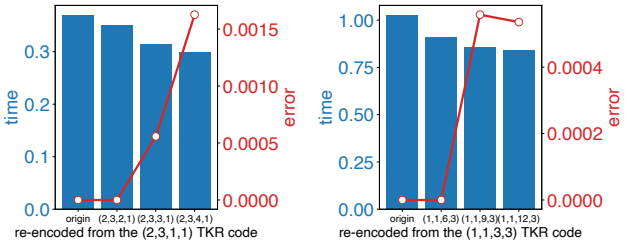


Fig. 7: The tradeoff between completion time and the numerical stability

We now demonstrate how local re-encoding for TKR codes helps to achieve a flexible tradeoff between completion time and the error before and after local re-encoding. This time we multiply two matrices of sizes 1024×7168 and 7168×1536, respectively. We launch two jobs that are encoded with $(2, 3, 1, 1)$ TKR code and $(1, 1, 3, 3)$ TKR code, respectively. Each job runs in a cluster of 9 workers and 1 master on Microsoft Azure. We then locally re-encode such two jobs with λ_p being 2, 3, and 4, respectively. Hence, we can expect to see the completion time reduced due to the lower complexity of the task and the higher error, illustrated in Fig. 7.

In Fig. 7, we can see that the completion time and the errors of the two jobs change as expected. With λ_p increasing, the completion time of the two jobs is saved by 19.1% and 18.3% eventually. On the other hand, the errors are also increased from 0 to 1.6×10^3 and 5.1×10^4. Hence, with local re-encoding, we don't need to change the number of workers while flexibly choosing between lower completion time or better numerical precision in the result.

VII. Discussions

As $\tilde{A}(x)$ and $\tilde{B}(x)$ are smaller with a larger m, n, or p, it is then impossible to re-encode them into larger matrices directly. In order to support bidirectional changes of the values of parameters, we can deploy $\tilde{A}(x)$ and $\tilde{B}(x)$ encoded with an EP code or a TKR code whose m, n, and p are small enough. The values of such parameters do not necessarily equal their actual values in the job, as we can always run local re-encoding before the job starts. In other words, whenever a job needs to run, the tasks can always be locally re-encoded to update the coding scheme and/or its parameters, especially when the performance of some resource has significantly changed. As shown in Fig. 4b and Fig. 5, the overhead of local re-encoding is marginal, so the job completion time will be close to that when coded tasks are originally encoded this way without re-encoding.

In order to maintain the tolerance against stragglers, it is also required that the number of tasks is sufficient even if the parameters are changed. Hence, the tasks also need to be deployed on a sufficient number of servers even though they are more than the number of workers required by the initial values of parameters, especially when their initial values are set small for the two-directional re-encoding. In this way, when a job is launched, a subset of such servers will be selected as workers so that the required number of stragglers can be tolerated. This will increase the overhead of encoding and storage, due to the additional tasks that may not run eventually. However, as coded tasks do not need to run on workers before the job starts, they will only be stored on potential workers and only selected workers will eventually run their tasks. Hence, the additional overhead will only be incurred from encoding and storage, not in the job.

Moreover, if coded tasks are initially stored in a distributed storage system, they will be typically stored on different storage servers, and we can store initially encoded tasks on such servers. Therefore, even if computation is separated from storage, we can still let each worker workers download one of such tasks directly and apply local re-encoding to get the desired coding scheme and parameters. Therefore, compared with encoding all coded tasks from scratch before the job start, local re-encoding significantly saves time and network bandwidth to deploy coded tasks to workers. The additional resource needed is only the storage space for storing coded tasks when the input matrices are stored, as the price for the flexibility.

VIII. Conclusion

As resources in the distributed infrastructure are shared by multiple jobs, their performances fluctuate with time dynamically. Although coded matrix multiplication has been demonstrated to tolerate stragglers, existing coding techniques do not support a flexible change of the coding scheme or parameters without receiving additional data. In this paper, we propose a framework of local re-encoding, which allows changing the coding scheme and/or its parameters for distributed matrix multiplication without incurring additional traffic. We demonstrate that our framework can significantly save the time and communication overhead to complete distributed matrix multiplication, and flexibly achieve the tradeoff between computation and communication, and that between completion time and numerical precision.

Acknowledgements

This paper is based upon work supported by the National Science Foundation (CCF-2101388). X. Zhong was supported by the Science and Technology Project of the Department of Education of Jiangxi Province, China (170384). J. Parker was supported by an NSF REU site program at the Florida International University (CNS-1851890).

References

[1] X. Su, X. Zhong, X. Fan, and J. Li, "Local Re-encoding for Coded Matrix Multiplication," in *IEEE International Symposium on Information Theory (ISIT)*, 2020, pp. 221–226.

[2] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray Failure: The Achilles' Heel of Cloud-Scale Systems," in *USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2017.

[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, 2018.

[4] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *International Conference on Machine Learning (ICML)*, 2017, pp. 3368–3376.

[5] V. Gupta, D. Carrano, Y. Yang, V. Shankar, T. Courtade, and K. Ramchandran, "Serverless Straggler Mitigation using Error-Correcting Codes," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 135–145.

[6] S. Dutta, V. Cadambe, and P. Grover, ""Short-Dot": Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," *IEEE Transactions on Information Theory*, vol. 65, no. 10, pp. 6171–6193, 2019.

[7] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," *IEEE Transactions on Information Theory*, vol. 66, no. 3, pp. 1920–1933, 2020.

[8] ——, "Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication," *Advances in Neural Information Processing Systems (NIPS)*, pp. 4406–4416, 2017.

[9] 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.

[10] S. Hong, H. Yang, and J. Lee, "Squeezed Polynomial Codes: Communication-Efficient Coded Computation in Straggler-Exploiting Distributed Matrix Multiplication," *IEEE Access*, vol. 8, pp. 190 516–190 528, 2020.

[11] M. Fahim and V. R. Cadambe, "Numerically Stable Polynomially Coded Computing," *IEEE Transactions on Information Theory*, vol. 67, no. 5, pp. 2758–2785, 2021.

[12] W. Gautschi, "How (Un)stable are Vandermonde Systems?" *Asymptotic and Computational Analysis*, 1990.

[13] L. Tang, K. Konstantinidis, and A. Ramamoorthy, "Erasure Coding for Distributed Matrix Multiplication for Matrices With Bounded Entries," *IEEE Communications Letters*, vol. 23, no. 1, pp. 8–11, jan 2019.

[14] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones," in *Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 185–198.

[15] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[16] N. B. Shah, K. Lee, and K. Ramchandran, "When Do Redundant Requests Reduce Latency?" *IEEE Transactions on Communications*, vol. 64, no. 2, pp. 715–722, 2016.

[17] Z. Qiu and J. F. Pérez, "Evaluating Replication for Parallel Jobs: An Efficient Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2288–2302, 2016.

[18] D. Wang, G. Joshi, and G. Wornell, "Efficient Task Replication for Fast Response Times in Parallel Computation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 599–600, 2014.

[19] K. Lee, R. Pedarsani, and K. Ramchandran, "On Scheduling Redundant Requests with Cancellation Overheads," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1279–1290, 2017.

[20] A. Mallick, M. Chaudhari, U. Sheth, G. Palanikumar, and G. Joshi, "Rateless Codes for Near-perfect Load Balancing in Distributed Matrix-vector Multiplication," in *2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, 2020, pp. 95–96.

[21] T. Baharav, K. Lee, O. Ocal, and K. Ramchandran, "Straggler-proofing Massive-scale Distributed Matrix Multiplication with d-dimensional Product Codes," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 1993–1997.

[22] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, "OverSketch: Approximate Matrix Multiplication for the Cloud," in *IEEE International Conference on Big Data*, 2018.

[23] H. Park, K. Lee, J.-Y. Sohn, C. Suh, and J. Moon, "Hierarchical Coding for Distributed Computing," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 1630–1634.

[24] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional Coded Matrix Multiplication," in *IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2418–2422.

[25] S. Kiani, N. Ferdinand, and S. C. Draper, "Exploitation of stragglers in coded computation," in *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 1988–1992.

[26] A. B. Das and A. Ramamoorthy, "Coded sparse matrix computation schemes that leverage partial stragglers," *IEEE Transactions on Information Theory*, vol. 68, no. 6, pp. 4156–4181, 2022.

[27] F. Maturana, V. S. C. Mukka, and K. V. Rashmi, "Access-optimal Linear MDS Convertible Codes for All Parameters," in *2020 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2020, pp. 577–582.

[28] F. Maturana and K. V. Rashmi, "Convertible Codes: Enabling Efficient Conversion of Coded Data in Distributed Storage," *IEEE Transactions on Information Theory*, vol. 68, no. 7, pp. 4392–4407, 2022.

[29] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *11th European PVM/MPI Users' Group Meeting*, 2004, pp. 97–104.

[30] L. Reichel and G. Opfer, "Chebyshev-Vandermonde Systems," *Mathematics of Computation*, vol. 57, no. 196, pp. 703–721, 1991.

[31] G. H. Golub and C. F. Van Loan, *Matrix Computations*. JHU press, 2013, vol. 3.

Xian Su is currently a Ph.D. student at the Graduate Center of the City University of New York. He received his B.S. degree from the College of Information Science and Engineering, Ocean University of China, China, in 2016, and was a master and Ph.D. student at Florida International University from 2018 to 2020. His research interests focus on coded matrix multiplication and distributed machine learning.

Jared Parker is currently attending Virginia Commonwealth University in pursuit of a bachelor's degree in Computer Science with minors in Mathematics and Computer Engineering. His research contributions were sponsored by the Research Experience for Undergraduates (REU) program held by Florida International University in Summer 2020.

Xiaomei Zhong is currently a Ph.D. student at the School of Electrical and Automation Engineering, East China Jiaotong University. She received her B.S. and M.S. degrees from the School of Computer Science, East China University of Technology, in 2001 and 2004, respectively. She is also serving as a lecturer at the School of Software, East China Jiaotong University. Her research interests include information security and the software formal method.

Xiaodi Fan received his B.S. degree in Communication Engineering at the Hangzhou Dianzi University in July 2018. He is currently a Ph.D. student at the Graduate Center of the City University of New York. His Ph.D. research is on designing and deploying new erasure code schemes to tolerate stragglers, and on leveraging the power of stragglers in large-scale distributed matrix multiplication and machine learning.

Jun Li received his Ph.D. degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2017, and his B.S. and M.S. degrees from the School of Computer Science, Fudan University, China, in 2009 and 2012. He is currently an assistant professor at the Queens College and Graduate Center, City University of New York. His research interests fall into the intersection between coding theory and distributed systems.