Dynamic Coding for Distributed Matrix Multiplication

Xian Su
Florida International University
Miami, FL 33199
xsu002@fiu.edu

Xiaodi Fan
Florida International University
Miami, FL 33199
xfan009@fiu.edu

Jun Li
Florida International University
Miami, FL 33199
junli@cs.fiu.edu

Abstract

Matrix multiplication is a fundamental operation in many machine learning algorithms. With the size of the dataset increasing rapidly, it is now a common practice to compute large-scale matrix multiplication on multiple servers, such that each server multiplies submatrices inside the input matrices. As straggling servers are inevitable in a distributed infrastructure, various coding schemes have been proposed which deploy coded tasks encoded from the submatrices of input matrices. The overall result can then be decoded from a subset of such coded tasks. However, as the resources are shared with other jobs in a distributed infrastructure and their performance can change dynamically, the optimal way to encode the input matrices may also change over time. So far, existing coding schemes for matrix multiplication all require to split the input matrices and encode them in advance, and cannot change the coding schemes or adjust their parameters after encoding. In this paper, we propose a coding framework that can dynamically change the coding schemes and their parameters, by only re-encoding local data in each task. We demonstrate that the original tasks can be quickly converted into new tasks only incurring marginal overhead.

1 Introduction

Matrix multiplication is an essential building block in various machine learning algorithms. With the growing size of the dataset, it is now common that the input matrices are too large to calculate the multiplication on a single server. Therefore, it becomes common to run such algorithms on multiple servers in a distributed infrastructure, *e.g.*, in a cloud, where each server executes a task multiplying two smaller matrices. However, it is well known that servers in a distributed infrastructure are not reliable, and are subject to various faulty behaviors [1]. For example, servers may experience temporary performance degradation, due to load imbalance or resource congestion [2]. Therefore, when distributing computation onto multiple servers, the progress of the algorithm 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, to multiply $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$ with B, *i.e.*, $AB = \begin{bmatrix} A_1B \\ A_2B \end{bmatrix}$, we can replicate each of the two tasks, *i.e.*, A_1B and A_2B, on multiple servers. This method, however, suffers from a high resource consumption. Only r stragglers can be tolerated when all tasks are replicated

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

$r+1$ times. On the other hand, coding-based approaches for distributed matrix multiplication have been proposed to tolerate stragglers more efficiently, where each server multiplies coded submatrices of A or/and B, *e.g.*, $(A_1 + A_2)B$. Therefore, if we run this coded task with A_1B and A_2B, we can recover the two submatrices in AB if any two of the three tasks are finished. Compared to replicating each task on two servers, this coding scheme can tolerate the same number of stragglers with 25% fewer tasks.

So far, existing coding schemes for matrix multiplication include polynomial codes [3], MatDot codes [4], and entangled polynomial codes [5], *etc*. These three coding schemes support to split the input matrices differently and thus achieve different recovery thresholds, *i.e.*, the number of tasks required to recover the overall result. The two coded matrices in each task must be encoded from A and B in advance before multiplication. However, the best coding scheme (and the values of its parameters) for a job of large-scale matrix multiplication usually depends on the resources such as CPU and network bandwidth. For example, if CPU is the bottleneck, it is desirable to split A and B into more submatrices. On the other hand, if the network bandwidth is limited, it becomes desirable to complete the computation on fewer tasks. Unfortunately, the performances of resources in a cloud are subject to change due to the shared nature of resources in the cloud. In this paper, we propose a framework that supports to dynamically change the coding schemes and their parameters by only locally re-encoding the coded matrices in each task, *i.e.*, without receiving any additional data. We demonstrate that polynomial codes and MatDot codes can be converted into entangled polynomial codes and our framework can support a flexible adjustment of their parameters in general.

2 Motivating Examples

We now present that the coding scheme of a task can be dynamically changed with local re-encoding. Specifically, tasks encoded with polynomial codes or MatDot codes can be converted into tasks encoded with entangled polynomial codes. We demonstrate the re-encoding of tasks with toy examples, and then present the general framework in Sec. 3.

Conversion from polynomial codes to entangled polynomial codes. We assume that a job of $A \times B$ has been encoded with a polynomial codes, where A and B are split into 2 submatrices vertically and horizontally, respectively. In other words, $A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$, and $B = \begin{bmatrix} B_0 & B_1 \end{bmatrix}$. Hence, a coded task can be encoded as the multiplication of $\tilde{A}_{\text{P}}$ and $\tilde{B}_{\text{P}}$, where $\tilde{A}_{\text{P}} = A_0\delta^0 + A_1\delta^2$ and $\tilde{B}_{\text{P}} = B_0\delta^0 + B_1\delta^1$. Here the value of δ should be unique in each task. Hence, $\tilde{A}_{\text{P}}\tilde{B}_{\text{P}} = A_0B_0\delta^0 + A_0B_1\delta^1 + A_1B_0\delta^2 + A_1B_1\delta^3$, which is a polynomial of δ with a degree of 3. As the value of δ in each task is unique, we can recover A_iB_j, $0 \le i, j \le 1$, which are the coefficients of such a polynomial, by interpolation from any four such tasks.

If we further split A_i and B_i as $A_i = \begin{bmatrix} A_{i,0} & A_{i,1} \end{bmatrix}$ and $B_i = \begin{bmatrix} B_{0,i} \\ B_{1,i} \end{bmatrix}$, $i = 0, 1$, then A and B are both vertically and horizontally split into a total of four submatrices. An entangled polynomial code can then be applied which encodes A and B into $\tilde{A}_{\text{EP}} = A_{0,0}\delta^0 + A_{0,1}\delta^1 + A_{1,0}\delta^4 + A_{1,1}\delta^5$ and $\tilde{B}_{\text{EP}} = B_{1,0}\delta^0 + B_{0,0}\delta^1 + B_{1,1}\delta^2 + B_{0,1}\delta^3$. Hence, $\tilde{A}_{\text{EP}}\tilde{B}_{\text{EP}}$ is a polynomial of δ with a degree of 8, *i.e.*, it can be interpolated with 9 tasks with different values of δ. The four submatrices in AB, *i.e.*, $A_{i,0}B_{0,j} + A_{i,1}B_{1,j}$, $0 \le i, j \le 1$, can then be found as coefficients of $\delta^1, \delta^3, \delta^5$, and δ^7, respectively.

If we need to change the coding scheme from polynomial codes to entangled polynomial codes, traditionally we can only encode A and B again from scratch with entangled polynomial codes, consuming a significant amount of time and network bandwidth to deploy the new coded matrices. However, we can see that $\tilde{A}_{\text{P}}$ can be horizontally split into two partitions, *i.e.*, $\begin{bmatrix} A_{0,0}\delta^0 + A_{1,0}\delta^2 & A_{0,1}\delta^0 + A_{1,1}\delta^2 \end{bmatrix}$. Similarly, $\tilde{B}_{\text{P}}$ can also be vertically split into two partitions $\begin{bmatrix} B_{0,0}\delta^0 + B_{0,1}\delta^1 \\ B_{1,0}\delta^0 + B_{1,1}\delta^1 \end{bmatrix}$. Hence, we can re-encode them as $(A_{0,0}\delta^0 + A_{1,0}\delta^2) + (A_{0,1}\delta^0 + A_{1,1}\delta^2)\delta^{0.5} = A_{0,0}\sigma^0 + A_{1,0}\sigma^4 + A_{0,1}\sigma^1 + A_{1,1}\delta^5$ where $\sigma = \delta^{0.5}$, and $(B_{0,0}\delta^0 + B_{0,1}\delta^1)\delta^{0.5} + (B_{1,0}\delta^0 + B_{1,1}\delta^1) = B_{0,0}\sigma^1 + B_{0,1}\sigma^3 + B_{1,0}\sigma^0 + B_{1,1}\sigma^2$. If δ is unique in each task and is already chosen to be positive, we can see that σ is also unique, and thus the task after re-encoding is equivalent as the task encoded with an entangled polynomial codes.

Conversion from MatDot codes to entangled polynomial codes. Different from polynomial codes, MatDot codes split A and B horizontally and vertically, respectively. In other words, $A = [A_0 \quad A_1]$, and $B = \begin{bmatrix} B_0 \\ B_1 \end{bmatrix}$. MatDot codes encode A and B as $\tilde{A}_{\text{MD}} = A_0\delta^0 + A_1\delta^1$ and $\tilde{B}_{\text{MD}} = B_1\delta^0 + B_1\delta^1$. Then $\tilde{A}_{\text{MD}}\tilde{B}_{\text{MD}}$ is a polynomial of δ with a degree of 2, where $AB = A_0B_0 + A_1B_1$ appears as the coefficient of δ^1.

Similarly, if we further split A_i vertically as $\begin{bmatrix} A_{0,i} \\ A_{1,i} \end{bmatrix}$, and B_i horizontally as $[B_{i,0} \quad B_{i,1}]$, $i = 0, 1$. We can then split $\tilde{A}_{\text{MD}}$ vertically as $\begin{bmatrix} A_{0,0}\delta^0 + A_{0,1}\delta^1 \\ A_{1,0}\delta^0 + A_{1,1}\delta^1 \end{bmatrix}$, and $\tilde{B}_{\text{MD}}$ horizontally as $\begin{bmatrix} B_{1,0}\delta^0 + B_{0,0}\delta^1 & B_{1,1}\delta^0 + B_{0,1}\delta^1 \end{bmatrix}$. Hence, we can also re-encode them as $(A_{0,0}\delta^0 + A_{0,1}\delta^1) + (A_{1,0}\delta^0 + A_{1,1}\delta^1)\delta^4$ and $(B_{1,0}\delta^0 + B_{0,0}\delta^1) + (B_{1,1}\delta^0 + B_{0,1}\delta^1)\delta^2$, which are equivalent as entangled polynomial codes.

3 General Results

In fact, polynomial codes can be seen as a special case of entangled polynomial codes with $p = 1$ and MatDot codes can be seen as a special case of entangled polynomial codes with $m = n = 1$. Therefore, the motivating examples can be generalized as a framework that flexibly adjust the parameters of entangled polynomial codes.

Beyond the motivating examples, we consider a more general case where the two input matrices A and B can be equally divided into mp and np submatrices, *i.e.,* $A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,p-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,p-1} \end{bmatrix}$, and $B = \begin{bmatrix} B_{0,0} & \cdots & B_{0,n-1} \\ \vdots & \ddots & \vdots \\ B_{p-1,0} & \cdots & B_{p-1,n-1} \end{bmatrix}$. The example in Sec. 2 corresponds to the case of $m = n = p = 2$. In general, $\tilde{A}_{\text{EP}}$ and $\tilde{B}_{\text{EP}}$ are encoded from the mp and np submatrices in A and B, respectively, leading to a recovery threshold of $pmn + p - 1$ with an (m, n, p) entangled polynomial code.[1] In this paper, we propose a dynamic framework (details can be found in Appendix B) that achieve the following property:

Theorem 1 *A task encoded with an (m, n, p) entangled polynomial code can be locally re-encoded into a task encoded with a $(\lambda_m m, \lambda_n n, \lambda_p p)$ entangled polynomial code, where λ_m, λ_n, and λ_p are positive integers.*

From this theorem, we can see that if a job is originally encoded with an (m, n, p) entangled polynomial code, we can further split and encode its $\tilde{A}_{\text{EP}}$ and $\tilde{B}_{\text{EP}}$, such that the new tasks are encode with a $(\lambda_m m, \lambda_n n, \lambda_p p)$ entangled polynomial codes, without obtaining any additional data from remote servers. Saving the complexity of each tasks by $\lambda_m\lambda_n\lambda_p$ times by increasing the recovery threshold to $\lambda_m m\lambda_n n\lambda_p p + \lambda_p p - 1$, our framework achieves a tradeoff between computation and communication overhead.[2] As it is not necessary for each task to obtain any additional data as the coded matrices in the new tasks can be directly re-encoded from $\tilde{A}_{\text{EP}}$ and $\tilde{B}_{\text{EP}}$, leading to a marginal overhead of re-encoding (experiment result can be found in Appendix D).

4 Conclusion

Although coded matrix multiplication has been demonstrated to tolerate stragglers, existing coding techniques all require fixed parameters and cannot flexibly adjust the coding schemes or even the values of their parameters. However, in a distributed infrastructure resources are known to be shared and unreliable, making the optimal values of parameters change over time. In this paper, we propose a dynamic framework for entangled polynomial codes which can change the values of parameters locally on each server without incurring additional traffic, and thus significantly save the time and communication overhead to complete the matrix multiplication with new parameters.

[1] Interested readers may find more details of entangled polynomial codes in Appendix A.

[2] Detailed analysis of complexity can be found in Appendix C.

References

[1] 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.

[2] 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.

[3] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication," *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[4] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," Tech. Rep., 2018. [Online]. Available: https://arxiv.org/pdf/1801.10292.pdf

[5] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 2022–2026.

[6] "Open MPI: Open Source High Performance Computing," p. 2019. [Online]. Available: https://www.open-mpi.org

A Preliminaries

Assume that the input matrices A and B are split into $m \times p$ and $p \times n$ submatrices. With entangled polynomial codes, if there are T servers in total, each server runs a task that calculates $\tilde{C}_i(m,n,p) = \tilde{A}_i(m,n,p) \cdot \tilde{B}_i(m,n,p)$, $1 \le i \le T$. In particular, $\tilde{A}_i(m,n,p) = \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,z}\delta_i^{pnx+z}$, and $\tilde{B}_i(m,n,p) = \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{p-1-z,y}\delta_i^{py+z}$. Therefore, we have $\tilde{C}_i(m,n,p) = \sum_{x=0}^{m-1}\sum_{y=0}^{n-1}\sum_{t=0}^{2p-2}\left(\sum_{l=\max\{0,t-p+1\}}^{\min\{p-1,t\}} A_{x,l}B_{p-1-t+l,y}\right)\delta_i^{pnx+py+t}$. We can see that each $\tilde{C}_i(m,n,p)$ is a polynomial function of δ_i whose degree is $pmn+p-2$. Therefore, we can solve the coefficients of δ_i with any $pmn+p-1$ out of the T tasks, as long as they have different values of δ_i. In this paper, we assume that T is always larger or equal to the recovery threshold. In other words, at most $T-(pmn+p-1)$ stragglers can be tolerated.

In particular, we can find that the coefficient of δ_i^{p-1} in $\tilde{C}_i(m,n,p)$ is $\sum_{l=0}^{p-1} A_{x,l}B_{l,y}$, $0 \le x \le m-1$, $0 \le y \le n-1$. Therefore, we can obtain the mn submatrices in AB after decoding.

For example, when $m=2, n=1, p=2$, we have $A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix}$, and $B = \begin{bmatrix} B_0 \\ B_1 \end{bmatrix}$. With entangled polynomial codes, we have $\tilde{C}_i(2,1,2) = (A_{0,0}{\delta_i}^0 + A_{0,1}{\delta_i}^1 + A_{1,0}{\delta_i}^2 + A_{1,1}{\delta_i}^3)(B_1\delta^0 + B_0\delta^1)$. If we have five tasks finished with different values of δ_i, $i=0,\ldots,4$, we will get

$$\begin{bmatrix} \delta_0^0 & \delta_0^1 & \delta_0^2 & \delta_0^3 & \delta_0^4 \\ \delta_1^0 & \delta_1^1 & \delta_1^2 & \delta_1^3 & \delta_1^4 \\ \delta_2^0 & \delta_2^1 & \delta_2^2 & \delta_2^3 & \delta_2^4 \\ \delta_3^0 & \delta_3^1 & \delta_3^2 & \delta_3^3 & \delta_3^4 \\ \delta_4^0 & \delta_4^1 & \delta_4^2 & \delta_4^3 & \delta_4^4 \end{bmatrix} \begin{bmatrix} A_{0,0}B_1 \\ A_{0,0}B_0 + A_{0,1}B_1 \\ A_{0,1}B_0 + A_{1,0}B_1 \\ A_{1,1}B_1 + A_{1,0}B_0 \\ A_{1,1}B_0 \end{bmatrix}. \tag{1}$$

The matrix on the left in (1) is a Vandermonde matrix which is invertible, and thus we can decode it by multiplying its inverse on the left or by Gaussian elimination. After decoding, we will be able to get $AB = \begin{bmatrix} A_{0,0}B_0 + A_{0,1}B_1 \\ A_{1,1}B_1 + A_{1,0}B_0 \end{bmatrix}$.

When $p=1$, the corresponding code becomes polynomial codes whose recovery threshold is mn. When $m=n=1$, the corresponding code becomes MatDot codes whose recovery threshold is $2p-1$.

B Re-encoding of Entangled Polynomial Codes (Proof of Theorem 1)

B.1 Changing p to $\lambda_p p$

We first show that a task with an (m,n,p) entangled polynomial code can be converted into a task with an $(m,n,\lambda_p p)$ entangled polynomial code. We first assume that the two input matrices A and B can be divided as:

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,\lambda_p p-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,\lambda_p p-1} \end{bmatrix}, \text{ and } B = \begin{bmatrix} B_{0,0} & \cdots & B_{0,n-1} \\ \vdots & \ddots & \vdots \\ B_{\lambda_p p-1,0} & \cdots & B_{\lambda_p p-1,n-1} \end{bmatrix}.$$

Although it is not necessary for an (m,n,p) entangled polynomial code to split A horizontally and B vertically into $\lambda_p p$ partitions, it is required by the $(m,n,\lambda_p p)$ entangled polynomial code after conversion. Therefore, the task encoded by the (m,n,p) entangled polynomial code can be written as

$$\begin{aligned} \tilde{A}_i(m,n,p) &= \sum_{x=0}^{m-1}\sum_{z=0}^{p-1}\begin{bmatrix} A_{x,\lambda_p z} & \cdots & A_{x,\lambda_p z+\lambda_p-1} \end{bmatrix}\delta_i^{pnx+z} \\ &= \begin{bmatrix} \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,\lambda_p z}\delta_i^{pnx+z} & \cdots & \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,\lambda_p z+\lambda_p-1}\delta_i^{pnx+z} \end{bmatrix} \\ &\triangleq \begin{bmatrix} \tilde{A}_{i,0}(p,m,n) & \cdots & \tilde{A}_{i,\lambda_p-1}(p,m,n) \end{bmatrix}, \end{aligned} \tag{2}$$

and

$$\tilde{B}_i(m,n,p) = \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} \begin{bmatrix} B_{(p-1-z)\lambda_p,y} \\ \vdots \\ B_{(p-1-z)\lambda_p+\lambda_p-1,y} \end{bmatrix} \delta_i^{py+z}$$
$$= \begin{bmatrix} \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{(p-1-z)\lambda_p,y}\delta_i^{py+z} \\ \vdots \\ \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{(p-1-z)\lambda_p+\lambda_p-1,y}\delta_i^{py+z} \end{bmatrix} \triangleq \begin{bmatrix} \tilde{B}_{i,0}(p,m,n) \\ \vdots \\ \tilde{B}_{i,\lambda_p-1}(p,m,n) \end{bmatrix}. \quad (3)$$

Now we are going to re-encode $\tilde{A}_i(m,n,p)$ and $\tilde{B}_i(m,n,p)$ into $\tilde{A}_i(m,n,\lambda_p p)$ and $\tilde{B}_i(m,n,\lambda_p p)$, respectively.

First, we define $\delta_i = \sigma_i^{\lambda_p}$. Then $\tilde{A}_{i,l}$ and $\tilde{B}_{i,l}$ can be rewritten as $\tilde{A}_{i,l}(p,m,n) = \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,\lambda_p z+l}\sigma_i^{(pnx+z)\lambda_p}$, and $\tilde{B}_{i,l}(p,m,n) = \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{(p-1-z)\lambda_p+l,y}\sigma_i^{(py+z)\lambda_p}$, $l = 0,\ldots,\lambda_p-1$.

We now re-encode $\tilde{A}_i(m,n,p)$ and $\tilde{B}_i(m,n,p)$ as

$$\sum_{l=0}^{\lambda_p-1}\tilde{A}_{i,l}(p,m,n)\sigma_i^l = \sum_{l=0}^{\lambda_p-1}\sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,\lambda_p z+l}\sigma_i^{(pnx+z)\lambda_p+l} = \sum_{x=0}^{m-1}\sum_{z=0}^{\lambda_p p-1} A_{x,z}\sigma_i^{\lambda_p pnx+z}$$
$$= \tilde{A}_i(m,n,\lambda_p p), \quad (4)$$

and

$$\sum_{l=0}^{\lambda_p-1}\tilde{B}_{i,\lambda_p-1-l}(p,m,n)\sigma_i^l = \sum_{l=0}^{\lambda_p-1}\sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{(p-1-z)\lambda_p+\lambda_p-1-l,y}\sigma_i^{(py+z)\lambda_p+l}$$
$$= \sum_{y=0}^{n-1}\sum_{z=0}^{\lambda_p p-1} B_{\lambda_p p-1-z,y}\sigma_i^{\lambda_p py+z} = \tilde{B}_i(m,n,\lambda_p p). \quad (5)$$

We can see that in (4) and (5), we construct the $\tilde{A}_i(m,n,\lambda_p p)$ and $\tilde{B}_i(m,n,\lambda_p p)$ from data in $\tilde{A}_i(m,n,p)$ and $\tilde{B}_i(m,n,p)$, respectively.

B.2 Changing m to $\lambda_m m$

Now we show that a task with an (m,n,p) entangled polynomial code can be locally converted into a task with a (λ_m,n,p) entangled polynomial code.

For simplicity, we assume that A can be vertically split into $\lambda_m m$ partitions. In other words,

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,p-1} \\ \vdots & \ddots & \vdots \\ A_{\lambda_m m-1,0} & \cdots & A_{\lambda_m m-1,p-1} \end{bmatrix}$$

Hence, we have

$$\tilde{A}_i(m,n,p) = \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} \begin{bmatrix} A_{\lambda_m x,z} \\ \vdots \\ A_{\lambda_m x+\lambda_m-1,z} \end{bmatrix} \delta_i^{pnx+z} = \begin{bmatrix} \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{\lambda_m x,z}\delta_i^{pnx+z} \\ \vdots \\ \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{\lambda_m x+\lambda_m-1,z}\delta_i^{pnx+z} \end{bmatrix}$$
$$\triangleq \begin{bmatrix} \tilde{A}_{i,0}(p,m,n) \\ \vdots \\ \tilde{A}_{i,\lambda_m-1}(p,m,n) \end{bmatrix}. \quad (6)$$

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

$$\sum_{l=0}^{\lambda_m-1}\tilde{A}_{i,l}(p,m,n)\delta^{lpmn} = \sum_{l=0}^{\lambda_m-1}\sum_{x=0}^{m-1}\sum_{z=0}^{p-1}A_{\lambda_m x+l,z}\delta_i^{pnx+z+lpmn}$$
$$= \sum_{l=0}^{\lambda_m-1}\sum_{x=0}^{m-1}\sum_{z=0}^{p-1}A_{\lambda_m x+l,z}\delta_i^{pn(lm+x)+z} = \tilde{A'}_i(\lambda_m m,n,p).$$

Here, after re-encoding, we generate $\tilde{A'}_i(\lambda_m m,n,p)$ which is encoded by a $(\lambda_m m,n,p)$ entangled polynomial code from a matrix A' with rows in A switched:

$$A' = \left[\begin{array}{ccc} A_{0,0} & \cdots & A_{0,p-1} \\ A_{\lambda_m,0} & \cdots & A_{\lambda_m,p-1} \\ \vdots & \vdots & \vdots \\ A_{\lambda_m(m-1),0} & \cdots & A_{\lambda_m(m-1),p-1} \\ \hdashline A_{1,0} & \cdots & A_{1,p-1} \\ \vdots & \vdots & \vdots \\ A_{\lambda_m(m-1)+1,0} & \cdots & A_{\lambda_m(m-1)+1,p-1} \\ \hdashline \vdots & \vdots & \vdots \\ \hdashline A_{\lambda_m-1,0} & \cdots & A_{\lambda_1,p-1} \\ \vdots & \vdots & \vdots \\ A_{\lambda_m(m-1)+\lambda_m-1,0} & \cdots & A_{\lambda_m(m-1)+\lambda_m-1,p-1} \end{array}\right].$$

Although the sequence of rows in A is switched, it will not change the result after decoding, since the sequence of rows in AB can be switched back in the same way.

B.3 Changing n to $\lambda_n n$

Similarly, we alao assume that B can be horizontally split into $\lambda_n n$ partitions, *i.e.*,

$$B = \begin{bmatrix} B_{0,0} & \cdots & B_{0,\lambda_n n-1} \\ \vdots & \ddots & \vdots \\ B_{p-1,0} & \cdots & B_{p-1,\lambda_n n-1} \end{bmatrix}.$$

The matrix B can then be encoded by an (m,n,p) entangled polynomial code as follows.

$$\begin{aligned}\tilde{B}_i(m,n,p) &= \sum_{y=0}^{n-1}\sum_{z=0}^{p-1}\left[B_{p-1-z,\lambda_n y} \quad \cdots \quad B_{p-1-z,\lambda_n y+\lambda_n-1}\right]\delta_i^{py+z} \\ &= \left[\textstyle\sum_{y=0}^{n-1}\sum_{z=0}^{p-1}B_{p-1-z,\lambda_n y}\delta_i^{py+z} \quad \cdots \quad \sum_{y=0}^{n-1}\sum_{z=0}^{p-1}B_{p-1-z,\lambda_n y+\lambda_n-1}\delta_i^{py+z}\right] \\ &\triangleq \left[\tilde{B}_{i,0}(p,m,n) \quad \cdots \quad \tilde{B}_{i,\lambda_n-1}(p,m,n)\right]. \end{aligned} \tag{7}$$

When we change n to λ_n, we also need to re-encode $\tilde{B}_i(m,n,p)$ only as

$$\sum_{l=0}^{\lambda_n-1}\tilde{B}_{i,l}(p,m,n)\delta^{lpmn} = \sum_{l=0}^{\lambda_n-1}\sum_{y=0}^{n-1}\sum_{z=0}^{p-1}B_{p-1-z,\lambda_n y+l}\delta_i^{py+z+lpmn}$$

Although it cannot be directly written as $\tilde{B}_i(m, \lambda_n n, p)$, we show that it is equivalent as an $(m, \lambda_n n, p)$ entangled polynomial code, as they achieve the same recovery threshold. Since

$$\begin{aligned}
&\tilde{A}_i(m,n,p) \cdot \left(\sum_{l=0}^{\lambda_n - 1} \sum_{y=0}^{n-1} \sum_{z=0}^{p-1} B_{p-1-z,\lambda_n y+l} \delta_i^{py+z+lpmn} \right) \\
&= \left(\sum_{x=0}^{m-1} \sum_{z=0}^{p-1} A_{x,z} \delta_i^{pnx+z} \right) \cdot \left(\sum_{l=0}^{\lambda_n - 1} \sum_{y=0}^{n-1} \sum_{z=0}^{p-1} B_{p-1-z,\lambda_n y+l} \delta_i^{py+z+lpmn} \right) \\
&= \sum_{x=0}^{m-1} \sum_{l=0}^{\lambda_n - 1} \sum_{y=0}^{n-1} \sum_{s=0}^{2p-2} \sum_{z=\max\{0,s-p+1\}}^{\min\{p-1,s\}} A_{x,z} B_{p-1-z,\lambda_n y+l} \delta_i^{pmnl+pnx+py+s} \qquad (8)
\end{aligned}$$

Therefore, the degree of the polynomial in (8) is $pmn(\lambda_n - 1) + pn(m-1) + p(n-1) + 2p - 2 = pm\lambda_n n + p - 1$, which is the same as that of an $(m, \lambda_n n, p)$ entangled polynomial code. Moreover, we can get the submatrices in AB, *i.e.*, $\sum_{z=0}^{p-1} A_{x,z} B_{p-1-z,\lambda_n y+l}$, when $s = p-1$.

B.4 Changing (m, n, p) to $(\lambda_m m, \lambda_n n, \lambda_p p)$

In general, when we need to change the values of the three parameters at the same time,[3] 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 $\tilde{A}_i(m,n,p)$ or $\tilde{B}_i(m,n,p)$. Rows in A are virtually shuffled when $\lambda_m \neq 1$. If $\lambda_n \neq 1$, neither $\tilde{A}_i(m, \lambda_n n, p)$ nor $\tilde{B}_i(m, \lambda_n n, p)$ is constructed exactly but they can maintain the recovery threshold of the corresponding entangled polynomial code. Therefore, we will first convert p to $\lambda_p p$, then m to $\lambda_m m$, and finally n to $\lambda_n n$.

Therefore, assume that each task is originally encoded with an (m, n, p) entangled polynomial code. If A and B are of size $\Lambda_m m \times \Lambda_p p$ and $\Lambda_p p \times \Lambda_n n$, then each task can be converted into any $(\lambda_m m, \lambda_n n, \lambda_p p)$ entangled polynomial code, as long as $\lambda_m | \Lambda_m$, $\lambda_n | \Lambda_n$, and $\lambda_p | \Lambda_p$. The more divisors Λ_m, Λ_n, and Λ_p have, the more entangled polynomial codes that we can convert to.

Moreover, even though λ_m/λ_n/λ_p is not a divisor of Λ_m/Λ_n/Λ_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 addtional 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 $\lambda_p \ll \Lambda_p$.

C Complexity Analysis

We now discuss the complexity of our framework, especially the complexity of its re-encoding, and compare it to the complexity of the original entangled polynomial code. We find that the complexity of re-encoding is marginal compared to the original complexity of the entangled polynomial code, which means that the overhead of re-encoding can be much cheaper than encoding all tasks again from scratch.

We analyze the complexity as the number of multiplication, since the overhead of addition is much cheaper than that of multiplication. In fact, as the code in this paper are linear, the scale of addition is the same to that of multiplication or smaller.

We assume the sizes of A and B are $M \times P$ and $P \times N$, and then the sizes of $\tilde{A}_i(m,n,p)$ and $\tilde{B}_i(m,n,p)$ 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) entangled polynomial code, both $\tilde{A}_i(m,n,p)$ and $\tilde{B}_i(m,n,p)$ are encoded as a linear combination of mp submatrices in A and np submatrices in B. Therefore, each element in A and B will be multiplied with a constant, and the complexity of $\tilde{A}$ and $\tilde{B}$ are $O(MP)$ and $O(NP)$, respectively. In

[3] Changing two paramters can be seen as a special case.

addition, the constants are powers of δ_i, 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 (m,n,p), the complexity of re-encoding is much lower. When p changes to $\lambda_p p$, $\tilde{A}$ and $\tilde{B}$ should be further split into λ_p partitions 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 $O(\frac{NP}{np})$. Therefore, 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 $\tilde{A}$ and $\tilde{B}$, the complexity of matrix multiplication in a task with an (m,n,p) entangled polynomial code is $\frac{MNP}{mnp}$. After re-encoding, the complexity of the matrix multiplication becomes $\frac{MNP}{\lambda_m m \lambda_n n \lambda_p p}$. Therefore, the complexity of a task, including re-encoding and matrix multiplication, is $O\left(\frac{P}{p}\left(\frac{M}{m}+\frac{N}{n}+\frac{MN}{\lambda_m m \lambda_n n \lambda_p}\right)\right) = O(\frac{MNP}{\lambda_m m \lambda_n n \lambda_p p})$ if M and N are much larger than λ_m and $\lambda_n n$, respectively. In other words, the complexity of re-encoding is also marginal to that of the task.

We can also find that the decoding overhead of the job will not be affected after conversion. Compared to a job with a $(\lambda_m m, \lambda_n n, \lambda_p p)$ entangled polynomial code, the decoding overhead of the dynamic entangled polynomial code after re-encoding will be the same, since the new code will be equivalent to a $(\lambda_m m, \lambda_n n, \lambda_p p)$ entangled polynomial code.

D Evaluation

We implement our design with OpenMPI [6], and run the jobs of matrix multiplications on a cluster of $T+1$ servers, where one server will run as a master and the other T servers will be used as workers. Initially, all tasks are encoded with an (m,n,p) entangled polynomial codes. The two input matrices A and B are encoded by the master, which then sends $\tilde{A}_i(m,n,p)$ and $\tilde{B}_i(m,n,p)$ to Worker i, $i=1,\ldots,T$. In each experiment, we assume that there will be no more than 5 stragglers to tolerate, and thus only a subset of workers will multiply the two coded matrices, and uploads the result to the master. The master, on the other hand, will keep polling if any new result has been received. It will stop receiving any new result once the number of results received reaches the corresponding recovery threshold, and terminate other unfinished tasks. In each experiment, due to the change of recovery threshold with different values of m, n, and p, we let $T=72$ such that the number of servers are always sufficient for any change of parameters in each experiment.

In our experiments, we focus on the case when the values of parameters need to be changed, and we compare two schemes. One is re-encoding by static entangled polynomial codes (static RE), which encodes all tasks again from scratch and sends $\tilde{A}_i(\lambda_m m, \lambda_n n, \lambda_p p)$ and $\tilde{B}_i(\lambda_m m, \lambda_n n, \lambda_p p)$ to Worker i, $i=1,\ldots,T$. The other is our framework, *i.e.*, dynamic re-encoding of entangled polynomial codes (dynamic RE), which directly re-encodes $\tilde{A}_i(m,n,p)$ and $\tilde{B}_i(m,n,p)$ on each local worker into $\tilde{A}_i(\lambda_m m, \lambda_n n, \lambda_p p)$ and $\tilde{B}_i(\lambda_m m, \lambda_n n, \lambda_p p)$, respectively. The rest of such two schemes are the same, *i.e.*, multiplying $\tilde{A}_i(\lambda_m m, \lambda_n n, \lambda_p p)$ and $\tilde{B}_i(\lambda_m m, \lambda_n n, \lambda_p p)$ on workers and uploading the result to the master. Note that although all T workers have the coded matrices for the corresponding task, which are used by re-encoding if necessary, we do not run all of them in each job in order to make a fair comparison. In fact, we fix the number of stragglers to tolerate as 5, such that the actual number of workers running equals the corresponding recovery threshold plus 5.

We now present our evaluation results running on a cluster of virtual machines hosted on Amazon EC2. We run the master on a virtual machine of type `t2.xlarge` and all workers on virtual machines of type `t2.small`. We set initial values of (m,n,p) as $(2,2,2)$, and encode input matrices of three jobs. The sizes of input matrices of such three jobs are shown in Fig. 1. In each job, we change the parameters with four configurations of $(\lambda_m, \lambda_n, \lambda_p)$: $(4,1,1)$, $(1,8,1)$, $(1,1,4)$, and $(2,2,2)$. In other words, we change the value of only one parameter in the first three configurations and change the values of all parameters in the last configuration.

We first compare the overhead of re-encoding in Fig. 2. With each configuration, we repeat each job 50 times and obtain the mean and standard deviation of its results. As for dynamic RE, the

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

Figure 1: Sizes of input matrices in the three jobs.

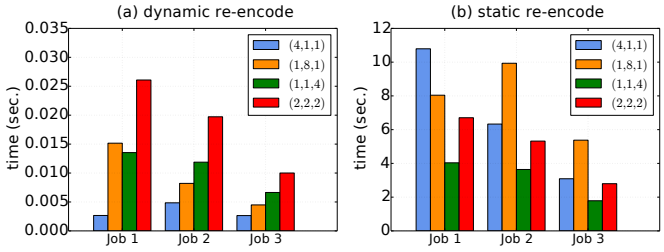

Figure 2: Overhead of re-encoding with different values of $(\lambda_m, \lambda_n, \lambda_p)$.

overhead of re-encoding comes only from re-encoding $\tilde{A}_i$ and $\tilde{B}_i$ locally. However, the static RE will be performed solely at the master, including the communication overhead of distributing all coded tasks as well. Therefore, although originally the time of static RE is between 1.79 seconds and 10.79 seconds, the dynamic RE only needs 0.026 seconds on average at most.

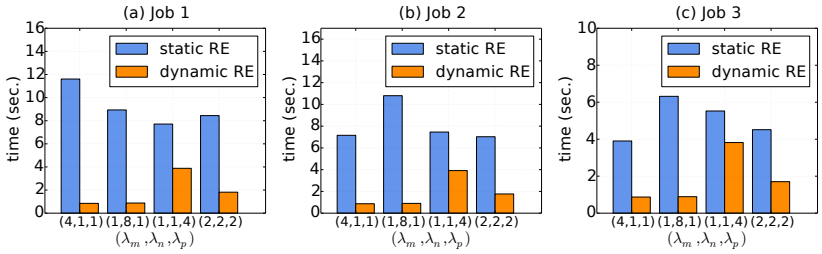

Figure 3: Job completion time with re-encoding.

Compared to the whole job completion time in Fig. 3, we can see that the overhead of dynamic RE in Fig. 2(a) is marginal. We also compare its job completion time with that of static RE. Due to the saved re-encoding overhead, we can observe that the job completion time can also be saved by up to 92.7%.