Local Re-encoding for Coded Matrix Multiplication

Xian Su*, Xiaomei Zhong†, Xiaodi Fan*, Jun Li*
*School of Computing and Information Sciences, Florida International University
†School of Software, East China Jiaotong University

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

I. 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 inevitable 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 can experience temporary performance degradation, due to load imbalance or resource congestion [1], [2], [3]. 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 the distributed matrix multiplication, a naive method is to replicate each task on multiple servers. For example, to multiply $A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$ with $B = \begin{bmatrix} B_0 & B_1 \end{bmatrix}$, *i.e.*, $AB = \begin{bmatrix} A_0B_0 & A_0B_1 \\ A_1B_0 & A_1B_1 \end{bmatrix}$, we can split the job into four tasks A_iB_j, $i \in [0,1]$ and $j \in [0,1]$, and replicate each of these four tasks on multiple servers. This method, however, suffers from high resource consumption. Only r stragglers can be tolerated when all tasks are replicated $r+1$ times. On the other hand, coding-based approaches for the distributed matrix multiplication have been proposed to tolerate stragglers more efficiently [2], [3], [4], [5], where each server multiplies coded submatrices of A or/and B, *e.g.*, $(A_1+A_2)(B_1+B_2)$. Therefore, if we run this coded task with the four original tasks, we can recover the four submatrices in AB if any four of the five tasks are finished. Compared to replicating each task on two servers, this coding scheme can tolerate any single straggler with 75% fewer additional tasks.

Existing coding schemes for the matrix multiplication include polynomial codes [6], MatDot codes [7], and entangled polynomial codes [5]. 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. Moreover, the two coded matrices in each task must be encoded in advance before the multiplication. However, the best coding scheme (and the values of its parameters) for a job of the 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.

To demonstrate the impact of resources on the performance of the distributed matrix multiplication, we run a job that multiplies two matrices of sizes 4096×4096, in our local cluster. The tasks in the job are encoded with a polynomial code, a MatDot code, and an entangled polynomial code, respectively. The job is implemented with Open MPI [8]. We calculate the results of tasks on servers with the same hardware configuration, called *workers*. Each worker uploads the result of each task to another server called *master*. The number of workers in the job is 5 more than the corresponding recovery threshold so that at most 5 stragglers can be tolerated. 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, the completion time of the job include the time of executing tasks on workers, uploading the results to the master, and decoding the results on the master.

In our experiment, we can observe how the performance of the job, in terms of its completion time, changes with network bandwidth. In order to change the 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 the 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. 1.

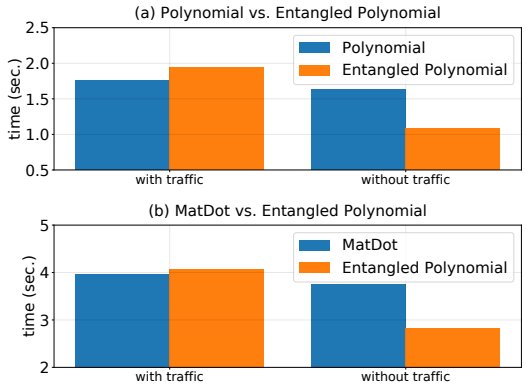

Fig. 1: Comparisons of completion time of the coded distributed matrix multiplication with and without additional traffic.

In Fig. 1a, we first compare the performance with a polynomial code and an 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. 1b, between a MatDot code and an 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 entangled polynomial codes can be more affected by the available bandwidth than the polynomial code and the MatDot code. This is because the task encoded by the entangled polynomial code has a lower complexity but the master also needs to receive more data before decoding. Furthermore, the three coding schemes also have different decoding complexities. If the CPU is shared by another job on the master or a worker, their completion time can also be affected differently. As the resource availability is subject to frequent changes in the cloud, it is challenging to choose the optimal coding scheme and parameters in advance. If we need to change the coding scheme, conventionally we can only encode tasks again from scratch, consuming a significant amount of time and network bandwidth to deploy the new coded matrices. A similar problem was investigated for distributed storage systems, where Maturana and Rashmi proposed *convertible codes* which allow changing the parameters of MDS codes with the optimal access cost [9], [10]. In this paper, we propose a framework for distributed matrix multiplication that supports changing 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 re-encoded into entangled polynomial codes and our framework can also support a flexible change of their parameters. In our experiments, we demonstrate that our framework can change the parameters within only 0.026 seconds at most. Compared to encoding tasks with updated parameters from scratch, our framework can save the overall completion time by up to 92.7%.

II. Preliminary

Assume that the input matrices A and B can be horizontally and vertically split into m and n submatrices, respectively. In other words, $A = \begin{bmatrix} A_0 \\ \vdots \\ A_{m-1} \end{bmatrix}$ and $B = \begin{bmatrix} B_0 & \cdots & B_{n-1} \end{bmatrix}$. In this case, a polynomial code can encode A and B into $\tilde{A}_{\text{P}}(m,n) = \sum_{x=0}^{m-1} A_x \delta^{nx}$ and $\tilde{B}_{\text{P}}(m,n) = \sum_{y=0}^{n-1} B_y \delta^{y}$, respectively. Hence, $\tilde{A}_{\text{P}}(m,n)\tilde{B}_{\text{P}}(m,n) = \sum_{x=0}^{m-1}\sum_{y=0}^{n-1} A_x B_y \delta^{nx+y}$ is a polynomial function of δ with a degree of $mn-1$, where $A_x B_y$, $x \in [0, m-1], y \in [0, n-1]$, appears as the coefficient of each term. Therefore, if we run tasks of $\tilde{A}_{\text{P}}(m,n)\tilde{B}_{\text{P}}(m,n)$ with different values of δ on multiple servers, we can recover all the coefficients from the results of any mn tasks by interpolation or Reed-Solomon decoding.

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 $B = \begin{bmatrix} B_0 \\ \vdots \\ B_{p-1} \end{bmatrix}$. Then $AB = \sum_{l=0}^{p-1} A_l B_l$. A MatDot code will encode A and B as $\tilde{A}_{\text{MD}}(p) = \sum_{z=0}^{p-1} A_z \delta^z$ and $\tilde{B}_{\text{MD}}(p) = \sum_{z=0}^{p-1} B_{p-1-z} \delta^z$, respectively. Hence, $\tilde{A}_{\text{MD}}(p)\tilde{B}_{\text{MD}}(p)$ is a polynomial function of δ with a degree of $2p-2$, whose coefficients can be decoded with any $2p-1$ tasks with different values of δ. For example, if $p = 2$, $\tilde{A}_{\text{MD}}(2) = A_0\delta^0 + A_1\delta^1$ and $\tilde{B}_{\text{MD}}(2) = B_1\delta^0 + B_0\delta^1$. Then their multiplication equals $A_0B_1\delta^0 + (A_0B_0 + A_1B_1)\delta^1 + A_1B_0\delta^2$. If we observe the coefficients, we can see that $\sum_{l=0}^{p-1} A_i B_i$ appears as the coefficient of δ^{p-1}. Therefore, the result of AB can be decoded as one of the coefficient of $\tilde{A}_{\text{MD}}(p)\tilde{B}_{\text{MD}}(p)$.

Now we assume that A and B are split into $m \times p$ and $p \times n$ submatrices, respectively. In other words, $A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,p-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,p-1} \end{bmatrix}$, and $B = \begin{bmatrix} B_{0,0} & \cdots & B_{0,n-1} \\ \vdots & \ddots & \vdots \\ B_{p-1,0} & \cdots & B_{p-1,n-1} \end{bmatrix}$. With an entangled polynomial code, each server runs a task that calculates $\tilde{A}_{\text{EP}}(m,n,p)\tilde{B}_{\text{EP}}(m,n,p)$, where $\tilde{A}_{\text{EP}}(m,n,p) = \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,z}\delta^{pnx+z}$, and $\tilde{B}_{\text{EP}}(m,n,p) = \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{p-1-z,y}\delta^{py+z}$. Hence,

$$\tilde{A}_{\text{EP}}(m,n,p)\tilde{B}_{\text{EP}}(m,n,p) = \sum_{x=0}^{m-1}\sum_{y=0}^{n-1}\sum_{t=0}^{2p-2}\left(\sum_{l=\max\{0,t-p+1\}}^{\min\{p-1,t\}} A_{x,l}B_{p-1-t+l,y}\right)\delta^{pnx+py+t}.$$

As $\tilde{A}_{\text{EP}}(m,n,p)\tilde{B}_{\text{EP}}(m,n,p)$ is a polynomial function of δ whose degree is $pmn+p-2$, we can solve the coefficients of δ with any $pmn+p-1$ tasks with different values of δ. In

particular, we can find that the coefficient of each term with $t = p-1$ is $\sum_{l=0}^{p-1} A_{x,l}B_{l,y}$, $0 \le x \le m-1$, $0 \le y \le n-1$. We can then obtain the mn submatrices in AB after decoding.

III. Coding Framework

In this paper, we present a framework that allows not only changing the coding schemes of a task with local re-encoding, but also changing the values of their parameters. In fact, we propose a framework 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 re-encode its $\tilde{A}_{\text{EP}}$ and $\tilde{B}_{\text{EP}}$, such that the new tasks are encoded with a $(\lambda_m m, \lambda_n n, \lambda_p p)$ entangled polynomial codes, without obtaining any additional data from remote servers, leading to a marginal overhead of re-encoding. Saving the complexity of each task 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 flexible tradeoff between computation and communication overhead.

The change of coding schemes can also be supported by this theorem. 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 $m = n = 1$. From Sec. II we can easily verify that $\tilde{A}_{\text{P}}(m, n) = \tilde{A}_{\text{EP}}(m, n, p = 1)$ and $\tilde{A}_{\text{MD}}(p) = \tilde{A}_{\text{EP}}(1, 1, p)$. The same equivalence can also be found in $\tilde{B}$. Hence, as a special case, Theorem 1 allows changing the coding schemes and also the parameters of a polynomial code or MatDot code. For convenience, we may omit EP in $\tilde{A}_{\text{EP}}(m, n, p)$ and $\tilde{B}_{\text{EP}}(m, n, p)$ in the rest of this paper if there is no ambiguity, *i.e.*, $\tilde{A}(m, n, p) = \tilde{A}_{\text{EP}}(m, n, p)$ and $\tilde{B}(m, n, p) = \tilde{B}_{\text{EP}}(m, n, p)$. We present the detailed framework in the rest of this section, which also proves Theorem 1.

A. Changing p to $\lambda_p p$

We first show that a task with an (m, n, p) entangled polynomial code can be locally re-encoded into a task with an $(m, n, \lambda_p p)$ entangled polynomial code. Assume that the two input matrices A and B can be divided as $A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,\lambda_p p-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,\lambda_p p-1} \end{bmatrix}$, 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 vertically and B horizontally into $\lambda_p p$ partitions, it is required by the $(m, n, \lambda_p p)$ entangled polynomial code after re-encoding. Therefore, the task encoded by the (m, n, p) entangled polynomial code can be written as

$$\begin{aligned}
&\tilde{A}(m, n, p) \\
&= \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} \left[A_{x,\lambda_p z} \ \cdots \ A_{x,\lambda_p z+\lambda_p-1}\right] \delta^{pnx+z} \\
&= \left[\sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,\lambda_p z}\delta^{pnx+z} \ \cdots \ \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,\lambda_p z+\lambda_p-1}\delta^{pnx+z}\right] \\
&\triangleq \left[\tilde{A}_0(m, n, p) \ \cdots \ \tilde{A}_{\lambda_p-1}(m, n, p)\right],
\end{aligned}$$

and

$$\begin{aligned}
\tilde{B}(m, n, p) &= \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} \begin{bmatrix} B_{(p-1-z)\lambda_p,y} \\ \vdots \\ B_{(p-1-z)\lambda_p+\lambda_p-1,y} \end{bmatrix} \delta^{py+z} \\
&= \begin{bmatrix} \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{(p-1-z)\lambda_p,y}\delta^{py+z} \\ \vdots \\ \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{(p-1-z)\lambda_p+\lambda_p-1,y}\delta^{py+z} \end{bmatrix} \\
&\triangleq \begin{bmatrix} \tilde{B}_0(m, n, p) \\ \vdots \\ \tilde{B}_{\lambda_p-1}(m, n, p) \end{bmatrix}.
\end{aligned}$$

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

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

$$\begin{aligned}
&\sum_{l=0}^{\lambda_p-1} \tilde{A}_l(m, n, p)\sigma^l \\
&= \sum_{l=0}^{\lambda_p-1}\sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,\lambda_p z+l}\sigma^{(pnx+z)\lambda_p+l} \\
&= \sum_{x=0}^{m-1}\sum_{z=0}^{\lambda_p p-1} A_{x,z}\sigma^{\lambda_p pnx+z} = \tilde{A}(m, n, \lambda_p p),
\end{aligned}$$

and

$$\begin{aligned}
&\sum_{l=0}^{\lambda_p-1} \tilde{B}_{\lambda_p-1-l}(m, n, p)\sigma^l \\
&= \sum_{l=0}^{\lambda_p-1}\sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{(p-1-z)\lambda_p+\lambda_p-1-l,y}\sigma^{(py+z)\lambda_p+l} \\
&= \sum_{y=0}^{n-1}\sum_{z=0}^{\lambda_p p-1} B_{\lambda_p p-1-z,y}\sigma^{\lambda_p py+z} = \tilde{B}(m, n, \lambda_p p).
\end{aligned}$$

B. Changing m to $\lambda_m m$

Now we show that a task with an (m, n, p) entangled polynomial code can be locally re-encoded into a task with a

$(\lambda_m m, n, p)$ entangled polynomial code. Assume that A can be horizontally split into $\lambda_m m$ partitions, *i.e.*,

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

Hence, we have

$$\tilde{A}(m,n,p) = \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} \begin{bmatrix} A_{\lambda_m x,z} \\ \vdots \\ A_{\lambda_m x+\lambda_m-1,z} \end{bmatrix} \delta^{pnx+z}$$

$$= \begin{bmatrix} \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{\lambda_m x,z}\delta^{pnx+z} \\ \vdots \\ \sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{\lambda_m x+\lambda_m-1,z}\delta^{pnx+z} \end{bmatrix} \triangleq \begin{bmatrix} \tilde{A}_0(m,n,p) \\ \vdots \\ \tilde{A}_{\lambda_m-1}(m,n,p) \end{bmatrix}.$$

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

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

Here, after re-encoding, we generate $\tilde{A}'(\lambda_m m,n,p)$ 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.

C. Changing n to $\lambda_n n$

Similarly, we also assume that B can be vertically 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:

$$\tilde{B}(m,n,p)$$
$$= \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} \begin{bmatrix} B_{p-1-z,\lambda_n y} & \cdots & B_{p-1-z,\lambda_n y+\lambda_n-1} \end{bmatrix} \delta^{py+z}$$
$$= \left[\sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{p-1-z,\lambda_n y}\delta^{py+z} \quad \cdots \quad \sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{p-1-z,\lambda_n y+\lambda_n-1}\delta^{py+z}\right]$$
$$\triangleq \begin{bmatrix} \tilde{B}_0(m,n,p) & \cdots & \tilde{B}_{\lambda_n-1}(m,n,p) \end{bmatrix}.$$

When we change n to $\lambda_n n$, we also need to re-encode $\tilde{B}(m,n,p)$ only as $\sum_{l=0}^{\lambda_n-1}\tilde{B}_l(m,n,p)\delta^{lpmn} = \sum_{l=0}^{\lambda_n-1}\sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{p-1-z,\lambda_n y+l}\delta^{py+z+lpmn}$.

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

$$\tilde{A}(m,n,p)\cdot\left(\sum_{l=0}^{\lambda_n-1}\sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{p-1-z,\lambda_n y+l}\delta^{py+z+lpmn}\right)$$
$$= \left(\sum_{x=0}^{m-1}\sum_{z=0}^{p-1} A_{x,z}\delta^{pnx+z}\right)\cdot$$
$$\left(\sum_{l=0}^{\lambda_n-1}\sum_{y=0}^{n-1}\sum_{z=0}^{p-1} B_{p-1-z,\lambda_n y+l}\delta^{py+z+lpmn}\right)$$
$$= \sum_{x=0}^{m-1}\sum_{l=0}^{\lambda_n-1}\sum_{y=0}^{n-1}\sum_{s=0}^{2p-2}\sum_{z=\max\{0,s-p+1\}}^{\min\{p-1,s\}} \Big(A_{x,z}\cdot B_{p-1-z,\lambda_n y+l}\delta^{pmnl+pnx+py+s}\Big),$$

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

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

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 $\tilde{A}(m,n,p)$ or $\tilde{B}(m,n,p)$. Rows in A are virtually shuffled when $\lambda_m \neq 1$. If $\lambda_n \neq 1$, neither $\tilde{A}(m,\lambda_n n,p)$ nor $\tilde{B}(m,\lambda_n n,p)$ is constructed exactly but they can maintain the recovery threshold of the corresponding entangled polynomial code. Therefore, we will first 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) 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 re-encoded into any $(\lambda_m m,\lambda_n n,\lambda_p p)$ entangled polynomial 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 entangled polynomial codes we can re-encode to.

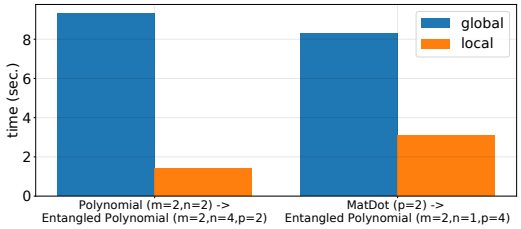

Fig. 2: Job completion time with re-encoding in the cluster.

Even if $\lambda_m/\lambda_n/\lambda_p$ is not a divisor of $\Lambda_m/\Lambda_n/\Lambda_p$, we can still add all-zero additional rows or columns into $\tilde{A}$ or/and $\tilde{B}$ so that they are divisible without affecting the overall result.

IV. Evaluation

We implement our framework with Open MPI based on the experiment in Sec. I. Before a worker executes the task, if the task needs to be re-encoded, the worker will first re-encode its $\tilde{A}$ and $\tilde{B}$, then multiply the two re-encoded matrices, and finally upload the result to the master. In other words, the completion time will also include the time of re-encoding. As for conventional framework (denoted by `global`), 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 re-encoding should include these three steps in `global`. The proposed `local` framework, on the other hand, allows re-encoding tasks into a new coding scheme on local directly.

We first present the results of running local re-encoding in our local cluster, with the same job and the same coding schemes in Sec. I. We still add 5 additional workers to tolerate at most 5 stragglers. In Fig. 2, we demonstrate the time of changing the coding schemes with the two schemes `global` and `local`. We repeat each configuration 50 times and show the average time. From Fig. 2, we can see that the time with `local` can be saved by 84.9% and 62.8%, respectively. The saving of time mainly comes from the saving of the time for encoding and deploying tasks with the new coding schemes, which can also be validated from the results running in Amazon EC2 below. Furthermore, if we compare the completion time of the job with the original coding schemes in Fig. 1, we can see that it can be saved by 14.3% and 18.5%, respectively.

In 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. 3. 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. 4. With each configuration, we also repeat each job 50 times and obtain the mean and standard deviation of its results. As for `local`,

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

Fig. 3: Sizes of input matrices in the three jobs.

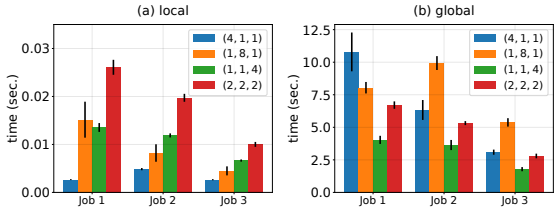

Fig. 4: Overhead of re-encoding with different values of $(\lambda_m, \lambda_n, \lambda_p)$.

the overhead of re-encoding comes only from re-encoding $\tilde{A}$ and $\tilde{B}$ locally. The overhead of `global`, however, comprises encoding which is performed solely at the master, and the overhead of distributing all coded tasks. Therefore, although originally the time of the re-encoding of `global` is between 1.79 seconds and 10.79 seconds, the re-encoding of `local` only needs 0.026 seconds on average at most.

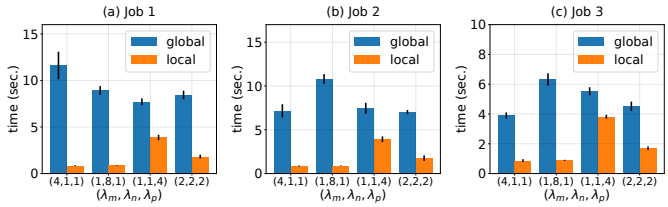

Fig. 5: Job completion time with re-encoding in Amazon EC2.

Compared to the job completion time in Fig. 5, we can see that the re-encoding overhead of `local` in Fig. 4a is marginal. We also compare its job completion time with that of `global`. Due to the saved re-encoding overhead, we can observe that the job completion time can also be saved by up to 92.7%.

V. Conclusion

Although the coded matrix multiplication has been demonstrated to tolerate stragglers, existing coding techniques cannot flexibly adjust the coding schemes or even the values of their parameters according to the change of resources in the distributed infrastructure. The proposed framework can change the coding schemes among representative coding schemes, as well as their parameters, for the distributed matrix multiplication without incurring additional traffic, and thus significantly save the time and communication overhead to complete the matrix multiplication with dynamic resources.

Acknowledgements

This paper is based upon work supported by the National Science Foundation (CCF-1910447), AWS Cloud Credits for Research, and the Science and Technology Project of the Department of Education of Jiangxi Province, China (170384).

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] 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.

[4] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *Advances in Neural Information Processing Systems*, 2016, pp. 2100–2108.

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

[7] 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

[8] 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.

[9] F. Maturana and K. V. Rashmi, "Convertible Codes: New Class of Codes for Efficient Conversion of Coded Data in Distributed Storage," in *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 151, 2020, pp. 66:1–66:26.

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