Aditya Ramamoorthy, Anindya Bijoy Das, and Li Tang

Straggler-Resistant Distributed Matrix Computation via Coding Theory

Removing a bottleneck in large-scale data processing

©ISTOCKPHOTO.COM/HAMSTER3D

Digital Object Identifier 10.1109/MSP.2020.2974149
Date of current version: 28 April 2020

The current big data era routinely requires the processing of large-scale data on massive distributed computing clusters. In these applications, data sets are often so large that they cannot be housed in the memory and/or the disk of any one computer. Thus, the data and the processing are typically distributed across multiple nodes. Distributed computation is thus a necessity rather than a luxury. The widespread use of such clusters presents several opportunities and advantages over traditional computing paradigms. However, it also presents newer challenges where coding-theoretic ideas have recently had a significant impact. Large-scale clusters (which can be heterogeneous in nature) suffer from the problem of stragglers, which are slow or failed worker nodes in the system. Thus, the overall speed of a computation is typically dominated by the slowest node in the absence of a sophisticated assignment of tasks to the worker nodes.

These issues are a potential bottleneck in several important and basic tasks, such as (but not limited to) the training of large-scale models in machine learning. Operations, such as matrix–vector multiplication and matrix–matrix multiplication (henceforth referred to as *matrix computations*), play a significant role in several parts of the machine learning pipeline [1] (see the "Applications of Matrix Computations Within Distributed Machine Learning" section). In this article, we review recent developments in the field of coding for straggler-resilient distributed matrix computations.

The conventional approach for tackling stragglers in distributed computation has been to run multiple copies of tasks on various machines [2], with the hope that at least one copy finishes on time. However, coded computation offers significant benefits for specific classes of problems. We illustrate this by means of a matrix–vector multiplication example in Figure 1. Consider the scenario where a user wants to compute $\mathbf{A}^T\mathbf{x}$, where $\mathbf{A}$ is a $t \times r$ matrix and $\mathbf{x}$ is a $t \times 1$ vector; both t and r are assumed to be large. The size of $\mathbf{A}$ precludes the possibility that the computation can take place on a single node. Accordingly, matrix $\mathbf{A}$ is block column decomposed as $\mathbf{A} = [\mathbf{A}_1\ \mathbf{A}_2\ \mathbf{A}_3]$, where each $\mathbf{A}_i$ is the same size. Each worker node is given the responsibility of computing two submatrix–vector products so that the

computational load on each worker is two-thirds of the original. We note here that the master node that creates the encoded matrices, e.g., $(\mathbf{A}_2 + \mathbf{A}_3)$, only needs to perform additions (and more generally scalar multiplications). The computationally intensive task of computing inner products of the rows of the encoded matrices with $\mathbf{x}$ is performed by the worker nodes. Even if one worker is a complete straggler, i.e., it fails, there is enough information for a master node to compute the final result. However, this requires the master node to solve a linear system of equations to decode the final result. A similar approach (with additional subtleties) can be used to arrive at a corresponding illustrative example for matrix–matrix multiplication.

Straggler mitigation using coding techniques has also been considered in a different body of work that broadly deals with reducing file-access delays when retrieving data from cloud storage systems [3]–[6]. Much of this article deals with understanding tradeoffs between file-access latency and the redundancy introduced by the coding method under different service time models for the servers within the cloud. Coded systems in turn introduce interesting challenges in the queuing delay analysis of these systems. These solutions can also be adapted for use within coded computing.

Applications of matrix computations within distributed machine learning

Computing high-dimensional linear transforms is an important component of dimensionality reduction techniques, such as principal component analysis and linear discriminant analysis [7]. Large-scale linear regression and filtering are also canonical examples of problems where linear transformations play a key role. They are also key components of training deep neural networks [1] and using them for classification, as we explain in detail.

Every layer of a fully connected deep neural network (see Figure 2) requires matrix–matrix multiplications in both forward and backward propagation. Suppose that the training data can be represented as a matrix $\mathbf{P}_0$ of size $f \times m$, where f is the number of features and m is the number of samples. In forward propagation, in any layer i the input $\mathbf{P}_{i-1}$ is multiplied by the weight matrix $\mathbf{W}_i$, and the bias term $\mathbf{b}_i$ is added. Following this, it is passed through a nonlinear function, $g_i(\cdot)$, to obtain $\mathbf{P}_i$ (the input of the next layer), i.e.,

$$\mathbf{Z}_i = \mathbf{W}_i\mathbf{P}_{i-1} + \mathbf{b}_i\mathbf{1}^T \quad \text{and} \quad \mathbf{P}_i = g_i(\mathbf{Z}_i).$$

We note here that if $\mathbf{W}_i$ is a large matrix, then we have a large-scale matrix–matrix multiplication problem that needs to be solved in this step.

Similar issues also arise in the backpropagation step, where the weight matrices and bias vectors are adjusted. We typically use a variant of gradient descent to obtain the weight matrix

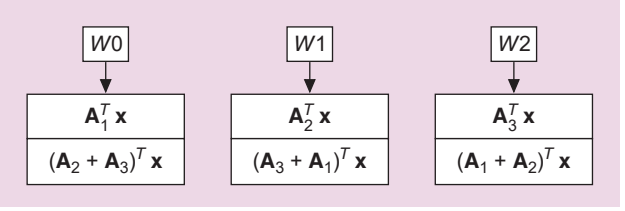

FIGURE 1. Matrix $\mathbf{A}$ is split into three equal-sized block columns. Each node is responsible for computing submatrix–vector products sequentially from top to bottom. Note that $\mathbf{A}^T\mathbf{x}$ can be decoded even if one node fails.

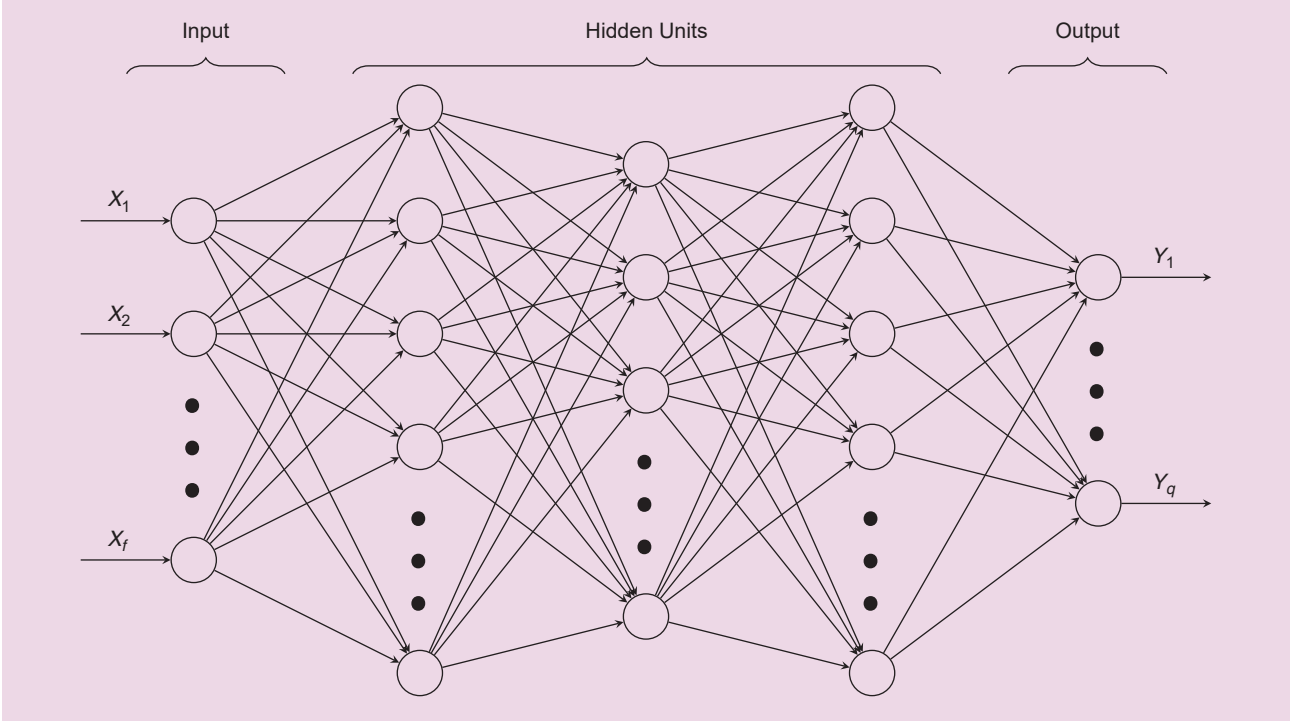

FIGURE 2. A fully connected neural network with three hidden layers where an input vector has a size f (number of features) and can be classified into one of q classes.

$\mathbf{W}_i^j$ at iteration j in layer i using an appropriate learning rate α. Now, if $d\mathbf{Z}_i^j$, $d\mathbf{W}_i^j$, and $d\mathbf{P}_i^j$ indicate the gradients of the chosen loss function with respect to $\mathbf{Z}_i$, $\mathbf{W}_i$, and $\mathbf{P}_i$ respectively, then for any iteration j, we compute

$$d\mathbf{Z}_i^j = g_i'(\mathbf{Z}_i^j) \odot d\mathbf{P}_i^j \quad \text{and} \quad d\mathbf{W}_i^j = \frac{1}{m} d\mathbf{Z}_i^j \mathbf{P}_{i-1}^{jT};$$

and update $\mathbf{W}_i^j = \mathbf{W}_i^{j-1} - \alpha d\mathbf{W}_i^j$ and $d\mathbf{P}_{i-1}^j = \mathbf{W}_i^{jT} d\mathbf{Z}_i^j$,

where the symbol $\odot$ indicates the Hadamard product. Thus, the backpropagation step requires matrix–matrix multiplication in each layer as well. Furthermore, each of these steps is repeated over multiple iterations.

As a concrete example, consider AlexNet [8], which performs a 1,000-way classification of the ImageNet data set and provides a top-five test error rate of under 15.3%. It has a training set of 1.2 million images, 50,000 validation images, and a test set of 150,000 images, each of which is a $224 \times 224 \times 3$ $(= 150,528)$ image. So, for training, $\mathbf{P}_0$ has a size $\approx 1.5 \times 10^5$ by 1.2×10^6. AlexNet consists of eight layers, among which five are convolutional layers and the other three are fully connected layers. Thus, this network has 43,264 and 4,096 neurons in the fifth and sixth layers. So, $\mathbf{W}_6$ has a size of $4{,}096 \times 43{,}264$. Thus, in the sixth layer of the forward propagation, the network requires the product of two matrices of size $4{,}096 \times 43{,}264$ and $43{,}264 \times (1.2 \times 10^6)$.

Problem formulation

We present a formulation of the distributed matrix–matrix multiplication problem in this section. Note that matrix–vector multiplication is a special (though very important) case of matrix–matrix multiplication, and the formulation carries over in this case in a natural manner. Consider a scenario where a master node has two large matrices, $\mathbf{A} \in \mathbb{R}^{t \times r}$ and $\mathbf{B} \in \mathbb{R}^{t \times w}$, and wishes to compute $\mathbf{A}^T\mathbf{B}$ in a distributed fashion using N worker nodes.

Each worker node is assigned a storage fraction for the coded columns of $\mathbf{A}$ (denoted γ_A) and $\mathbf{B}$ (denoted γ_B). The coded columns of $\mathbf{A}$ and $\mathbf{B}$ should be created by means of computationally inexpensive operations, e.g., scalar multiplications and additions. While the storage fraction constraint can be satisfied by potentially nonlinear coded solutions, our primary interest will be in linearly coded solutions where $\mathbf{A}$ and $\mathbf{B}$ are decomposed into block matrices of size $p \times m$ and $p \times n$ respectively, as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0,0} & \dots & \mathbf{A}_{0,m-1} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{p-1,0} & \dots & \mathbf{A}_{p-1,m-1} \end{bmatrix}, \text{ and } \mathbf{B} = \begin{bmatrix} \mathbf{B}_{0,0} & \dots & \mathbf{B}_{0,n-1} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{p-1,0} & \dots & \mathbf{B}_{p-1,n-1} \end{bmatrix}, \tag{1}$$

so that the blocks in $\mathbf{A}$ and $\mathbf{B}$ are of size $(t/p) \times (r/m)$ and $(t/p) \times (w/n)$, respectively. The master node generates certain linear combinations of the blocks in $\mathbf{A}$ and $\mathbf{B}$ and sends them to the worker nodes. The master node also requires each worker node to compute the product of some or all of their assigned matrices in a specified sequential order; we refer to this as the *responsibility* of the worker node. For instance, if a given worker node stores coded matrices $\tilde{\mathbf{A}}_0$, $\tilde{\mathbf{A}}_1$ and $\tilde{\mathbf{B}}_0$, $\tilde{\mathbf{B}}_1$ and is required to compute all four pairwise products, then the scheme specifies the order, e.g., $\tilde{\mathbf{A}}_0^T\tilde{\mathbf{B}}_0$, $\tilde{\mathbf{A}}_1^T\tilde{\mathbf{B}}_0$, $\tilde{\mathbf{A}}_0^T\tilde{\mathbf{B}}_1$, $\tilde{\mathbf{A}}_1^T\tilde{\mathbf{B}}_1$ or $\tilde{\mathbf{A}}_1^T\tilde{\mathbf{B}}_1$, $\tilde{\mathbf{A}}_1^T\tilde{\mathbf{B}}_0$, $\tilde{\mathbf{A}}_0^T\tilde{\mathbf{B}}_1$, $\tilde{\mathbf{A}}_0^T\tilde{\mathbf{B}}_0$, and so on. The following two cases of block-partitioning $\mathbf{A}$ and $\mathbf{B}$ are of special interest.

- *Case 1* $(p = 1)$: In this scenario, both $\mathbf{A}$ and $\mathbf{B}$ are decomposed into block columns, i.e., $\mathbf{A} = [\mathbf{A}_0\, \mathbf{A}_1 \dots \mathbf{A}_{m-1}]$ and $\mathbf{B} = [\mathbf{B}_0\, \mathbf{B}_1 \dots \mathbf{B}_{n-1}]$, so that recovering $\mathbf{A}^T\mathbf{B}$ is equivalent to recovering $\mathbf{A}_i^T\mathbf{B}_j$ for all pairs $i = 0, \dots m-1$, $j = 0, \dots, n-1$.
- *Case 2* $(m = n = 1)$: We set $\mathbf{A}^T = [\mathbf{A}_0^T\, \mathbf{A}_1^T \dots \mathbf{A}_{p-1}^T]$ and $\mathbf{B}^T = [\mathbf{B}_0^T\, \mathbf{B}_1^T \dots \mathbf{B}_{p-1}^T]$ so that $\mathbf{A}^T\mathbf{B} = \sum_{i=0}^{p-1} \mathbf{A}_i^T\mathbf{B}_i$.

The computational cost of computing $\mathbf{A}^T\mathbf{B}$ is $rw(2t-1)$ floating point operations (flops), which is approximately $\text{cost}(r, t, w) = 2rtw$ when t is large. In the distributed setup under consideration, the computational load on each worker node is less than the original cost of computing $\mathbf{A}^T\mathbf{B}$, and the advantages of parallelism can therefore be leveraged.

We note some minor differences in the matrix–vector multiplication scenario at this point. Here, the master node wishes to compute $\mathbf{A}^T\mathbf{x}$, where $\mathbf{x}$ is a vector. As $\mathbf{x}$ is much smaller than $\mathbf{A}$, we typically only impose the storage constraint for the worker nodes for the matrix $\mathbf{A}$ and assume that $\mathbf{x}$ is available to all of them. The case when $\mathbf{x}$ is further split into subvectors [9] will be treated along with the matrix–matrix multiplication case.

Example 1

Consider distributed matrix multiplication with $p = 1$ and $m = n = 2$. Furthermore, we define the matrix polynomials as follows:

$$\mathbf{A}(z) = \mathbf{A}_0 + \mathbf{A}_1 z \text{ and } \mathbf{B}(z) = \mathbf{B}_0 + \mathbf{B}_1 z^2;$$

$$\text{so that } \mathbf{A}^T(z)\mathbf{B}(z) = \mathbf{A}_0^T\mathbf{B}_0 + \mathbf{A}_1^T\mathbf{B}_0 z + \mathbf{A}_0^T\mathbf{B}_1 z^2 + \mathbf{A}_1^T\mathbf{B}_1 z^3.$$

Suppose that the master node evaluates $\mathbf{A}(z)$ and $\mathbf{B}(z)$ at distinct points $z_0, \dots, z_{N-1}$. It sends $\mathbf{A}(z_i)$ and $\mathbf{B}(z_i)$ to the ith worker node, which is assigned the responsibility of computing $\mathbf{A}^T(z_i)\mathbf{B}(z_i)$.

It follows that as soon as any four out of the N worker nodes return the results of their computation, the master node can perform polynomial interpolation to recover the (k, l)th entry of each $\mathbf{A}_i^T\mathbf{B}_j$ for $0 \le k < r/2$ and $0 \le l < w/2$. Therefore, such a system is resilient to $N - 4$ failures.

Note here that each worker node stores coded versions of $\mathbf{A}$ and $\mathbf{B}$ of size $t \times r/2$ and $t \times w/2$ respectively, i.e., $\gamma_A = \gamma_B = 1/2$. The computational load on each worker is $\text{cost}(r/2, t, w/2) = \text{cost}(r, t, w)/4$, i.e., one-fourth of the original. Furthermore, each worker communicates a $r/2 \times w/2$ matrix to the master node.

On the other hand, splitting the matrices as in case 2 yields a different tradeoff.

Example 2

Let $m = n = 1$ and $p = 2$, so that $\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} \mathbf{B}_0 \\ \mathbf{B}_1 \end{bmatrix}$ and consider the following matrix polynomials:

$$\mathbf{A}(z) = \mathbf{A}_0 z + \mathbf{A}_1 \quad \text{and} \quad \mathbf{B}(z) = \mathbf{B}_0 + \mathbf{B}_1 z,$$

$$\text{so that } \mathbf{A}^T(z)\mathbf{B}(z) = \mathbf{A}_1^T\mathbf{B}_0 + (\mathbf{A}_0^T\mathbf{B}_0 + \mathbf{A}_1^T\mathbf{B}_1)z + \mathbf{A}_0^T\mathbf{B}_1 z^2.$$

As before, the master node evaluates $\mathbf{A}(z)$ and $\mathbf{B}(z)$ at distinct points $z_0, \ldots, z_{N-1}$ and sends the coded matrices to the worker nodes that calculate $\mathbf{A}^T(z_i)\mathbf{B}(z_i)$. In this case, as soon as any three workers complete their tasks, the master node can interpolate to recover $\mathbf{A}^T(z)\mathbf{B}(z)$ and obtain the desired result $(\mathbf{A}_0^T\mathbf{B}_0 + \mathbf{A}_1^T\mathbf{B}_1)$ as the coefficient of z. The other coefficients are interference terms. Thus, this system is resilient to $N-3$ stragglers and strictly improves on Example 1 with the same storage fraction $\gamma_A = \gamma_B = 1/2$.

The dimensions of $\mathbf{A}(z)$ and $\mathbf{B}(z)$ are $t/2 \times r$ and $t/2 \times w$ so that the computational load on each worker is $\text{cost}(r, t/2, w) = \text{cost}(r, t, w)/2$, i.e., it is twice that of the workers in Example 1. Moreover, each worker node communicates an $r \times w$ matrix to the master node, i.e., the communication load is four times that of Example 1.

Metrics for evaluating coded computing solutions

Examples 1 and 2 illustrate the core metrics by which coded computing solutions are evaluated. More formally, for given storage fractions γ_A and γ_B and the responsibilities of all worker nodes, we evaluate a solution by a subset of the following metrics.

- *Recovery threshold*: We say that a solution has recovery threshold τ if $\mathbf{A}^T\mathbf{B}$ can be decoded by the master node as long as any τ worker nodes return the results of their computation, e.g., the thresholds were four and three respectively in Examples 1 and 2. This metric is most useful under the assumption that worker nodes are either working properly or in failure.
- *Recovery threshold(II)*: A more refined notion of recovery is required when we consider scenarios where worker nodes may be slow but not complete failures. For instance, Figure 3 shows an example where each worker node is assigned two matrix–vector products and operates sequentially from top to bottom. It can be verified by inspection that as long as any three matrix–vector products are obtained from the worker nodes in this manner, the master node has enough information to decode $\mathbf{A}^T\mathbf{x}$. For instance, Figure 3(a) depicts a situation where $W2$ is failed and $W0$ is slow as compared to $W1$. The solution leverages the partial computations of $W0$ as well. We say that a solution has a recovery threshold(II) of τ' if the master node can decode the intended result if it receives the result of τ' computations from the worker nodes; these computations have to respect the sequential order within each worker node.
- *Computational load per worker node*: The complexity of determining $\mathbf{A}^T\mathbf{B}$ is $\text{cost}(r, t, w)$ flops. The computational load per worker is measured as a fraction of $\text{cost}(r, t, w)$, e.g., in Examples 1 and 2, the fractions are 1/4 and 1/2, respectively. If $\mathbf{A}$ and $\mathbf{B}$ are sparse, then the computational load on the worker will depend on the number of nonzero entries in them. We discuss this point in more detail in the section "Opportunities for Future Work."
- *Communication load per worker node*: The communication load per worker measures the number of values that a worker node needs to send to the master node, normalized by rw.
- *Decoding complexity*: All linear schemes under consideration in this article require solving a system of linear equations to decode the result $\mathbf{A}^T\mathbf{B}$. The time-complexity of solving an arbitrary $\ell \times \ell$ system of equations grows as ℓ^3. This is another metric that needs to be small enough for a scheme to be useful. For instance, in Example 1, the master node needs to solve a 4×4 system of equations $rw/4$ times. Thus, the time cost of decoding is roughly proportional to rw; there is no dependence on t. On the other hand, the computation load on a worker depends in a multiplicative manner on t. In scenarios where t is large, it can be argued that the decoding cost is negligible compared to the worker computation. Nevertheless, this is a metric that needs to be considered. Decoding in Examples 1 and 2 corresponds to polynomial interpolation and is thus a "structured" system of equations that can be typically solved much faster than Gaussian elimination.
- *Numerical stability*: Solving linear equations to determine $\mathbf{A}^T\mathbf{B}$ naturally brings up the issue of numerical stability of the decoding. Specifically, if the system of equations is ill-conditioned, then the decoded result may suffer from significant numerical inaccuracies. Let $\mathbf{P}$ be a real-valued matrix and $\sigma_{\max}(\mathbf{P})$ and $\sigma_{\min}(\mathbf{P})$ denote its maximum and minimum singular values [10]. We define its condition number as $\text{cond}(\mathbf{P}) = \sigma_{\max}(\mathbf{P})/\sigma_{\min}(\mathbf{P})$.

As a rule of thumb, if the system of equations has a condition number of 10^l, it results in the loss of approximately l bits of numerical precision. For any distributed scheme, we ideally want the worst-case condition number over all possible recovery matrices to be as small as possible.

Overview of techniques

The overarching idea in almost all of the works in this area is one of "embedding" the matrix computation into the structure of an erasure code. Note that (n, k) erasure codes [11] used in point-to-point communication have the property that one can

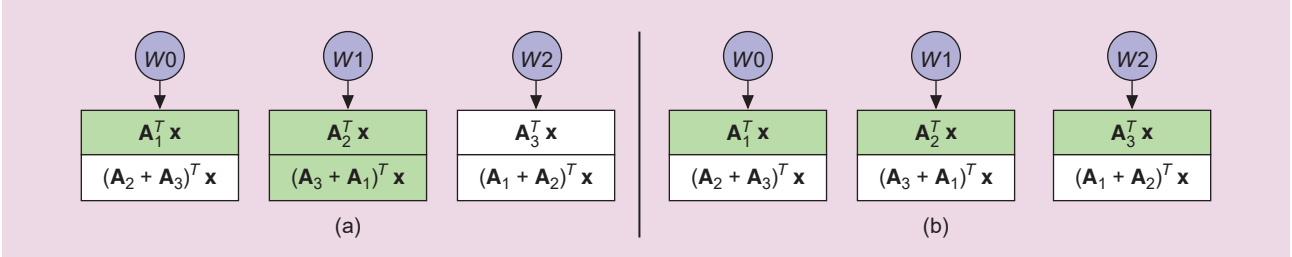

FIGURE 3. (a) and (b) Two example scenarios where the master node obtains the results of three completed tasks (respecting the sequential order) from the worker nodes. The scheme is such that the master node is guaranteed to recover $\mathbf{A}^T\mathbf{x}$ as long as any three tasks are completed.

decode the intended message for several erasure patterns, e.g., for maximum distance separable (MDS) codes, as long as any k coded symbols (out of n) are obtained, the receiver can decode the intended message. Most constructions of MDS codes are nonbinary and decoding typically involves multiplications and divisions. There are also several constructions of binary, near-MDS codes, e.g., low-density parity-check and fountain codes, that allow recovery with high probability from any $k(1+\epsilon)$ symbols for small $\epsilon > 0$ when n and k are large. The decoding can be performed by simple add/subtract operations.

For instance, Example 1 demonstrates an embedding of matrix–matrix multiplication into the structure of a Reed–Solomon code. This embedding essentially requires that the ith worker node computes the evaluation of polynomial $\mathbf{A}^T(z)\mathbf{B}(z)$ at z_i; this evaluation may or may not be received based on whether the ith worker node is a straggler. In contrast, in the traditional communication scenario, the transmitter computes the evaluation and the channel uncertainty dictates whether or not the evaluation is received. Moreover, the decoding in Example 1 corresponds to polynomial interpolation, which is precisely what Reed–Solomon decoding (from erasures) amounts to. Despite the similarities, we emphasize that, in the matrix computation setup, we operate within the real field $\mathbb{R}$, while traditional erasure coding almost exclusively considers operations over finite fields. As we will see, this introduces additional complications in the distributed computation scenario.

The original idea of using redundancy to protect against node failures in distributed matrix computation goes back to the work on "algorithm-based fault tolerance" from the 1980s [12], [13]. However, more recent contributions have significantly improved on them. Ideas from polynomial evaluation and interpolation have played an important role in this area. We briefly recapitulate some of these ideas in the following sections.

Primer on polynomials

Let $u(z) = \Sigma_{k=0}^{d} u_k z^k$ be a polynomial of degree d with real coefficients. Let $u^{(j)}(z)$ denote the jth derivative of $u(z)$. It can be verified that

$$u^{(j)}(z) = \sum_{k=0}^{d} u_k \binom{k}{j} j!\, z^{k-j}, \tag{2}$$

where $\binom{k}{i} = 0$ if $k < i$. Furthermore, note that we can also represent $u(z)$ by considering its Taylor series expansion around a point $\beta \in \mathbb{R}$, i.e.,

$$u(z) = \sum_{k=0}^{d} \frac{u^{(k)}(\beta)}{k!}(z-\beta)^k. \tag{3}$$

It is well known that $u(z)$ has a zero of multiplicity ℓ at $\beta \in \mathbb{R}$ if and only if $u^{(i)}(\beta) = 0$ for $0 \le i < \ell$ and $u^{(\ell)}(\beta) \neq 0$.

Another well-known fact states that if we obtain $d+1$ evaluations of $u(z)$ at distinct points $z_1, \ldots, z_{d+1}$, then we can interpolate to find the coefficients of $u(z)$. This follows from the fact that the Vandermonde matrix $\mathbf{V}$ with parameters $z_1, \ldots, z_{d+1}$, i.e., $\mathbf{V}_{i,j} = z_j^i$ for $0 \le i \le d, 1 \le j \le d+1$, is nonsingular when z_j, $j = 1, \ldots, d+1$ are distinct. An interesting generalization holds when we consider not only the evaluations of $u(z)$ but also its derivatives. We illustrate this by the following example.

Example 3

Let $d = 2$, so that $u(z) = u_0 + u_1 z + u_2 z^2$ and the first derivative $u^{(1)}(z) = u_1 + 2u_2 z$. Suppose we obtain $u(z_1)$, $u^{(1)}(z_1)$, and $u(z_2)$, where $z_1 \neq z_2$. We claim that this suffices to recover $u(z)$. To see this, assume otherwise, i.e., there exists $\tilde{u}(z) \neq u(z)$ such that $u(z_1) = \tilde{u}(z_1)$, $u^{(1)}(z_1) = \tilde{u}^{(1)}(z_1)$, and $u(z_2) = \tilde{u}(z_2)$. This in turn implies that there exists a polynomial $a(z) = u(z) - \tilde{u}(z)$ such that $a(z_1) = a^{(1)}(z_1) = a(z_2) = 0$. Now, we note that $a(z)$ is such that it has a zero of multiplicity 2 at z_1 and a zero of multiplicity 1 at z_2. The fundamental theorem of algebra states that if a polynomial has more zeros (counting multiplicities) than its degree, then it has to be identically zero. Therefore, we can conclude that $a(z)$ is identically zero, and we can recover $u(z)$ exactly. This can also be equivalently be seen by examining the following:

$$\det\begin{bmatrix} 1 & 0 & 1 \\ z_1 & 1 & z_2 \\ z_1^2 & 2z_1 & z_2^2 \end{bmatrix} = (z_2^2 - 2z_1 z_2) + (2z_1^2 - z_1^2) = (z_1 - z_2)^2 > 0.$$

In general, for a polynomial $u(z) = \Sigma_{k=0}^{d} u_k z^k$ of degree d, suppose that we obtain $u^{(\ell)}(z_i)$ for $\ell = 0, \ldots, k_i - 1$ [where $u^{(0)}(z) = u(z)$] for distinct points $z_i, i = 1, \ldots, N$. In this case, if $\Sigma_{i=1}^{N} k_i \ge d+1$, then we can recover $u(z)$ exactly [14].

Note that polynomial interpolation is equivalent to solving a Vandermonde system of equations. However, since this system of equations is structured, the complexity can be reduced. Specifically, a degree-d polynomial can be interpolated with time-complexity $O(d \log^2 d)$ [15].

Distributed matrix–vector multiplication

In more recent times, the power of coding-theoretic methods for matrix–vector multiplication was first explored in the work of Lee et al. [20]. In the notation of the section "Problem Formulation," set $p = 1$ and consider splitting $\mathbf{A} = [\mathbf{A}_0\ \mathbf{A}_1\ \ldots\ \mathbf{A}_{m-1}]$ into m equal-sized block columns. Here m is a parameter that is a design choice. The idea of Lee et al. [20] is to pick the generator matrix of a (N, m) MDS code denoted $\mathbf{G} = (g_{ij}) \in \mathbb{R}^{m \times N}$. The master node then computes

$$\tilde{\mathbf{A}}_l = \sum_{i=0}^{m-1} g_{il} \mathbf{A}_i$$

and distributes $\mathbf{x}$ and $\tilde{\mathbf{A}}_l$ to the lth worker node for $l = 0, \ldots, N-1$, which computes $\tilde{\mathbf{A}}_l^T \mathbf{x}$. The master node wishes to decode $\mathbf{A}_j^T \mathbf{x}$, $j = 0, \ldots m-1$. Suppose that worker nodes indexed by $i_0, \ldots, i_{m-1}$ are the first m nodes to return their results. Note that the master node has

$$\tilde{\mathbf{A}}_{i_l}^T \mathbf{x} = \sum_{j=0}^{m-1} g_{ji_l} (\mathbf{A}_i^T \mathbf{x}), \quad \text{for } l = 0, \ldots, m-1,$$

which implies that it can solve a system of linear equations to determine the required result if the $m \times m$ submatrix of $\mathbf{G}$ indexed by columns $i_0, \ldots, i_{m-1}$ is nonsingular; the MDS property

of $\mathbf{G}$ guarantees this. Typical choices of $\mathbf{G}$ include picking it as a Vandermonde matrix with distinct parameters $z_1, \ldots, z_N$.

In this case, each $\tilde{\mathbf{A}}_l$ is the evaluation of $\mathbf{A}(z) = \mathbf{A}_0 + \mathbf{A}_1 z + \cdots + \mathbf{A}_{m-1} z^{m-1}$ at $z = z_l$. The recovery threshold is m, the computational and communication load of each worker node is $1/m$th of the original and the decoding can be performed faster than Gaussian elimination (see the section "Primer on Polynomials"). However, numerical stability is a significant concern when $\mathbf{G}$ has the Vandermonde form. It is well known from the numerical analysis literature [21] that the condition number of a $\ell \times \ell$ real Vandermonde matrix grows exponentially in ℓ. Column 2 of Table 1 contains some illustrative figures. It shows that even for $N = 30$ with a threshold $\tau = 28$, the condition number is too high to be useful in practice.

On the other hand, choosing each entry of $\mathbf{G}$ independent identically distributed (i.i.d.) at random from a continuous distribution also works with high probability, and the computational load per worker is still $1/m$th of the original. Numerical stability is better [22]; however, decoding the system of equations will typically take time, which is cubic in the size of the system of equations.

One can also use the idea of using polynomial interpolation with multiplicities discussed in the section "Primer on Polynomials." Let the jth derivative of $\mathbf{A}(z)$ be defined as follows:

$$\mathbf{A}^{(j)}(z) = \sum_{k=0}^{m-1} \mathbf{A}_k \binom{k}{j} j!\, z^{k-j}.$$

Suppose that the storage fraction $\gamma_A = 2/m$. In this case, for the ith worker node, the master node assigns the computation of first $[\mathbf{A}(z_i)]^T \mathbf{x}$ and then $[\mathbf{A}^{(1)}(z_i)]^T \mathbf{x}$. As soon as a worker node completes a task, it sends the result to the master node. The result of Ramamoorthy et al. [16] demonstrates that as long as the master node receives m matrix–vector multiplication results, it can decode the intended result, i.e., its recovery threshold(II) is m. The computational and communication load of each worker is $2/m$th of the original. The key advantage of this scheme is that it allows the master node to leverage partial computations performed by slow nodes. However, numerical stability continues to be a problem here.

The numerical stability issue with both approaches discussed can be addressed (to a certain extent) by a related idea that involves polynomials over finite fields. In particular, one can define polynomials over finite fields and their corresponding Hasse derivatives (resulting in so-called universally decodable matrices) and use an isomorphism between finite field elements and appropriate matrices to arrive at "binary" schemes that have a much better behaved condition number. We illustrate the basic idea here and refer the reader to [16] for the full details.

Example 4

Let $u(z) = u_0 + u_1 z + u_2 z^2$ be a polynomial of degree 2. The discussion in the section "Primer on Polynomials" indicates that an associated 3×3 Vandermonde matrix is nonsingular when the polynomial is evaluated at distinct points z_1, z_2, and z_3. It turns out that we can instead evaluate the polynomial at appropriately defined matrices instead and obtain schemes with useful properties. Let binary matrix $\mathbf{C}$ correspond to the matrix representation of the finite field $GF(2^3)$ (see [16] and [23] for details) and consider powers of $\mathbf{C}$, i.e., $\mathbf{C}^\ell$ reduced modulo-2, as

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \text{ and so, e.g., } \mathbf{C}^2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \mod 2.$$

Consider the $\mathbf{G}$ specified as follows (where each power of $\mathbf{C}$ is reduced modulo-2):

$$\mathbf{G} = \begin{bmatrix} \mathbf{I} & \mathbf{I} & \mathbf{I} & \mathbf{I} \\ \mathbf{I} & \mathbf{C} & \mathbf{C}^2 & \mathbf{C}^3 \\ \mathbf{I} & \mathbf{C}^2 & \mathbf{C}^4 & \mathbf{C}^6 \end{bmatrix}.$$

The work of Ramamoorthy et al. [16] shows, for example, that any 3×3 block matrix of $\mathbf{G}$ is nonsingular. For instance, the 9×9 matrix formed by picking the first three block columns has determinant –1 over $\mathbb{R}$. In the matrix–vector multiplication scenario, we can use $\mathbf{G}$ as the coding matrix (see Figure 4) by setting $m = 9$. This system can tolerate one failure.

An advantage of this method is that $\mathbf{G}$ is binary. Moreover, it has a significantly better worst-case condition number as compared to the polynomial approach (see Table 1, column 4). However, we are unaware of efficient decoding techniques for these methods. Thus, the decoding complexity is equivalent to Gaussian elimination.

Convolutional codes are another class of erasure codes where messages are encoded into sequences of varying lengths. As an example, consider two row vectors in $\mathbb{R}^3$, $\mathbf{u}_0 = [u_{00}\ u_{01}\ u_{02}]$ and $\mathbf{u}_1 = [u_{10}\ u_{11}\ u_{12}]$. These vectors can also be represented as polynomials $\mathbf{u}_i(D) = \sum_{j=0}^{2} u_{ij} D^j$ for $i = 0, 1$, where D is an indeterminate. Consider the following encoding of $[\mathbf{u}_0(D)\ \ \mathbf{u}_1(D)]$:

Table 1. The worst-case condition numbers for the different schemes.

Scenario	Polynomial [17]*	Orthogonal Polynomial [18]	[16]+Embedding†	All-Ones [19]	Random [19]
$\mathbf{N = 15}, \boldsymbol{\tau = 13}$	1.689×10^6	686.27	411	910	264.49
$\mathbf{N = 15}, \boldsymbol{\tau = 12}$	1.695×10^6	7,612.7	949	1.066×10^4	1.111×10^3
$\mathbf{N = 30}, \boldsymbol{\tau = 28}$	2.293×10^{13}	7,902.6	–	2,868.32	1,374.59

*Vandermonde scheme: the parameters are spaced uniformly in [–1, 1].
†Each worker node is assigned two matrix–vector products corresponding to the polynomial evaluation and its first derivative. The embedding matrix $\mathbf{C}$ corresponds to the matrix representation of $GF(3^3)$.

$$[\mathbf{c}_0(D)\ \ \mathbf{c}_1(D)\ \ \mathbf{c}_2(D)\ \ \mathbf{c}_3(D)] = [\mathbf{u}_0(D)\,\mathbf{u}_1(D)]\begin{bmatrix}1 & 0 & 1 & 1\\ 0 & 1 & 1 & D\end{bmatrix}$$
$$= [\mathbf{u}_0(D)\ \ \mathbf{u}_1(D)\ \ (\mathbf{u}_0(D)+\mathbf{u}_1(D))\ \ (\mathbf{u}_0(D)+D\mathbf{u}_1(D))].$$

It is not too hard to see that the polynomials $\mathbf{u}_0(D)$ and $\mathbf{u}_1(D)$ (equivalently the vectors $\mathbf{u}_0, \mathbf{u}_1$) can be recovered from any two entries of the vector $[\mathbf{c}_0(D)\ \mathbf{c}_1(D)\ \mathbf{c}_2(D)\ \mathbf{c}_3(D)]$. For instance, suppose that we only have $\mathbf{c}_2(D)$ and $\mathbf{c}_3(D)$ where

$$\mathbf{c}_2(D) = (u_{00}+u_{10}) + (u_{01}+u_{11})D + (u_{02}+u_{12})D^2 \text{ and}$$
$$\mathbf{c}_3(D) = u_{00} + (u_{01}+u_{10})D + (u_{02}+u_{11})D^2 + u_{12}D^3.$$

Starting with u_{00} from the constant term of $\mathbf{c}_3(D)$, one can recover u_{10} from $\mathbf{c}_2(D)$ and iteratively u_{01} from $\mathbf{c}_3(D)$ and so on. A similar argument applies if we consider a different pair of entries from $[\mathbf{c}_0(D)\ \mathbf{c}_1(D)\ \mathbf{c}_2(D)\ \mathbf{c}_3(D)]$. Distributed matrix–vector multiplication can be embedded into this convolutional code by interpreting the coefficients of the powers of D as the assignments to the workers (see [19] and [24]).

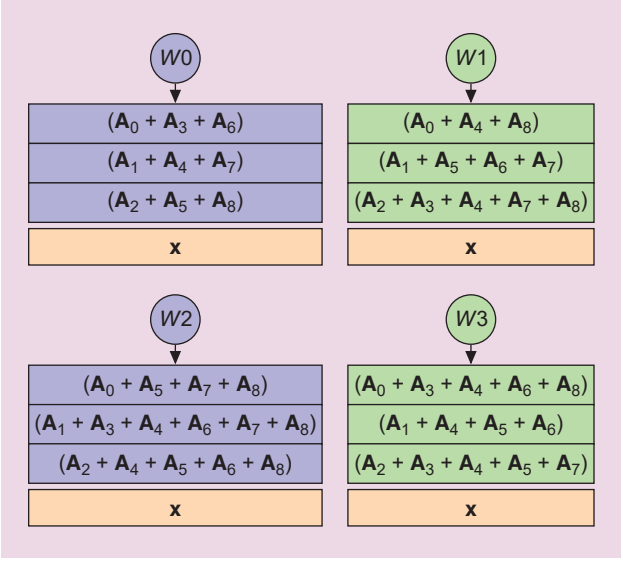

FIGURE 4. The scheme corresponding to the approach of Ramamoorthy et al. [16] as described in Example 4.

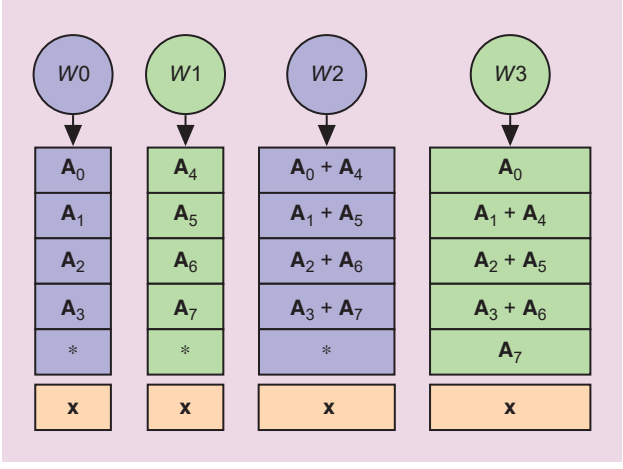

FIGURE 5. The scheme corresponding to the approach of Das et al. [19] as described in Example 5.

Example 5

Consider a system with $N = 4$ workers, with $\gamma_A = 5/8$. We partition $\mathbf{A}$ into $m = 8$ block columns of equal size, which are denoted as $\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A}_7$. So, we have $\mathcal{A}_0(D) = \mathbf{A}_0^T + \mathbf{A}_1^T D + \mathbf{A}_2^T D^2 + \mathbf{A}_3^T D^3$ and $\mathcal{A}_1(D) = \mathbf{A}_4^T + \mathbf{A}_5^T D + \mathbf{A}_6^T D^2 + \mathbf{A}_7^T D^3$. The matrices assigned to the ith worker are given by the coefficient of the powers of D in $\mathbf{C}_i(D)$, where

$$[\mathbf{C}_0(D)\ \ \mathbf{C}_1(D)\ \ \mathbf{C}_2(D)\ \ \mathbf{C}_3(D)] = [\mathcal{A}_0(D)\ \ \mathcal{A}_1(D)]\begin{bmatrix}1 & 0 & 1 & 1\\ 0 & 1 & 1 & D\end{bmatrix}.$$

This is illustrated in Figure 5. It can be verified that the system is resilient to two failures. Furthermore, it can be shown that the system of equations that the master node has to solve can be put in lower-triangular form upon appropriate permutations. Thus, decoding is quite efficient. This approach leads to a slightly nonuniform assignment of tasks to the different worker nodes, e.g., $W3$ has one additional matrix–vector product to compute as compared to the other worker nodes. However, this nonuniformity can be made as small as desired by choosing a large enough m, while ensuring that the decoding complexity remains low. It also has a much better condition number as compared to the polynomial-based schemes (see Table 1, column 5). It turns out that multiplying the elements of the encoding matrix by random numbers allows us to provide upper bounds on the worst-case condition number of the recovery matrices (see Table 1, column 6). Decoding in this case requires a least-squares solution; this least-squares solution can be made more efficient by exploiting the sparse nature of the underlying matrices [19].

A fountain coding approach (also known as *rateless coding*) was presented in the work of Mallick et al. [25]. In this scenario, the master node keeps computing random binary linear combinations of the $\mathbf{A}_i$s and sending them to the worker nodes. These combinations are chosen from a carefully designed degree sequence. The properties of this degree sequence guarantee with high probability that as long as the receiver obtains $m(1+\epsilon)$ matrix–vector products where $\epsilon > 0$ is a small constant, the receiver can decode the desired result (the result is asymptotic in m). Furthermore, this decoding can be performed using a so-called peeling decoder, which is much simpler than running full-blown Gaussian elimination. In a peeling decoder, at each time instant, the receiver can find one equation where there is only one unknown. This is important because, in the large m regime, the cubic complexity of Gaussian elimination would be unacceptably high, whereas the peeling decoder has a complexity $\approx m \log m$.

Distributed matrix–matrix multiplication

The situation is somewhat more involved when considering the distributed computation of $\mathbf{A}^T\mathbf{B}$. In this case, one needs to consider the joint design of the coded versions of the blocks of $\mathbf{A}$ and $\mathbf{B}$ [see (1)]. This topic was the focus of

the so-called algorithm-based fault-tolerance (ABFT) techniques [12], [13] in the 1980s. However, ABFT techniques result in suboptimal recovery thresholds. Yu et al. [17] presented an elegant solution to this problem based on polynomials. Their solution matches a corresponding lower bound on the threshold in certain cases. Interestingly, work on embedding matrix–matrix multiplication into the structure of polynomials was considered much earlier in the work of Yagle [26]; however, this was in the context of speeding up the computation rather than straggler resilience.

The basic ideas of using polynomials for matrix–matrix multiplication have already been illustrated by Examples 1 and 2 in the section "Problem Formulation." We now present a more in-depth discussion of these techniques along with a host of other approaches that have been considered in the literature. The first idea along these lines in Yu et al. [17] corresponds to the case of $p = 1$ and arbitrary m and n (using the notation introduced in the section "Problem Formulation"). As before, polynomial $\mathbf{A}(z) = \Sigma_{i=0}^{m-1} \mathbf{A}_i z^i$. However, the second polynomial with coefficients $\mathbf{B}_j$, $j = 0, \ldots, n-1$ needs to be chosen more carefully. The underlying simple and useful trick is to choose $\mathbf{B}(z)$ in such a way that $\mathbf{A}_i^T \mathbf{B}_j$ for $i = 0, \ldots, m-1$, $j = 0, \ldots, n-1$ appear as coefficients of z^l for $l = 0, \ldots, mn-1$ of the polynomial $\mathbf{A}^T(z)\mathbf{B}(z)$. Yu et al. [17] propose

$$\mathbf{A}(z) = \sum_{j=0}^{m-1} \mathbf{A}_j z^j \quad \text{and} \quad \mathbf{B}(z) = \sum_{j=0}^{n-1} \mathbf{B}_j z^{jm},$$

$$\text{so that } \mathbf{A}^T(z)\mathbf{B}(z) = \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} \mathbf{A}_j^T \mathbf{B}_k z^{j+km}.$$

The ith worker node is assigned $\mathbf{A}(z_i)$ and $\mathbf{B}(z_i)$ so that the storage fractions are $\gamma_A = 1/m$ and $\gamma_B = 1/n$. The node is tasked with computing $\mathbf{A}^T(z_i)\mathbf{B}(z_i)$. Evidently, $\mathbf{A}^T(z)\mathbf{B}(z)$ can be interpolated to determine the intended result as long as the master node obtains mn distinct evaluations of it. This solution is such that the computational load and the communication load on each worker is $1/mn$th of the original. It also achieves the optimal recovery threshold (under communication load limitations on the worker nodes). Furthermore, the decoding complexity corresponds to running rw/mn polynomial interpolations of a degree $(mn-1)$ polynomial. Nevertheless, this technique has serious numerical stability issues stemming from the ill-conditioned nature of the Vandermonde-structured recovery matrices discussed before (see the section "Distributed Matrix–Vector Multiplication").

A generalization of this approach for matrix–matrix multiplication when $p > 1$ was considered in Yu et al. [27] and Dutta et al. [28] around the same time. This was earlier examined in the matrix–vector context when each worker only gets subvectors of $\mathbf{x}$ in the work of Dutta et al. [9], which can be considered as a special case of this result when $n = 1$. However, the threshold in Yu et al. [27] is better than that in Dutta et al. [9]. Our discussion loosely follows the presentation in Yu et al. [27]. Note that when $p = 1$, our unknowns are precisely $\mathbf{A}_i^T \mathbf{B}_j$, $i = 0, \ldots, m-1$, $j = 0, \ldots, n-1$. However, when $p = 2$ (for instance), the unknowns now involve the sum of certain terms. Indeed, when $m = n = p = 2$, we have

$$\mathbf{A}^T\mathbf{B} = \begin{bmatrix} \mathbf{A}_{00}^T\mathbf{B}_{00} + \mathbf{A}_{10}^T\mathbf{B}_{10} & \mathbf{A}_{00}^T\mathbf{B}_{01} + \mathbf{A}_{10}^T\mathbf{B}_{11} \\ \mathbf{A}_{01}^T\mathbf{B}_{00} + \mathbf{A}_{11}^T\mathbf{B}_{10} & \mathbf{A}_{01}^T\mathbf{B}_{01} + \mathbf{A}_{11}^T\mathbf{B}_{11} \end{bmatrix}.$$

Recall, that our goal is to form polynomials $\mathbf{A}(z)$ and $\mathbf{B}(z)$ with coefficients from $\mathbf{A}_{ij}$, $i = 0, \ldots, m-1$, $j = 0, \ldots, p-1$ and $\mathbf{B}_{kl}$, $k = 0, \ldots, p-1$, $l = 0, \ldots, n-1$ such that the useful terms appear as appropriate coefficients of consecutive powers of z in $\mathbf{A}^T(z)\mathbf{B}(z)$. When $p > 1$ (unlike $p = 1$), the presence of interference terms becomes unavoidable. Nevertheless, one can choose $\mathbf{A}(z)$ and $\mathbf{B}(z)$ in such a way that we can interpolate the useful terms along with interference terms at the master node. This can lead to a strictly better threshold as indicated in Example 2; see [27] for full details. For $m = n = p = 2$, we choose

$$\begin{aligned} \mathbf{A}(z) &= \mathbf{A}_{00} + \mathbf{A}_{10} z + \mathbf{A}_{01} z^2 + \mathbf{A}_{11} z^3, \\ \mathbf{B}(z) &= \mathbf{B}_{10} + \mathbf{B}_{00} z + \mathbf{B}_{11} z^4 + \mathbf{B}_{01} z^5, \quad \text{so that} \\ \mathbf{A}^T(z)\mathbf{B}(z) &= (*) + (\mathbf{A}_{00}^T\mathbf{B}_{00} + \mathbf{A}_{10}^T\mathbf{B}_{10}) z + (*) z^2 \\ &\quad + (\mathbf{A}_{01}^T\mathbf{B}_{00} + \mathbf{A}_{11}^T\mathbf{B}_{10}) z^3 + (*) z^4 + (\mathbf{A}_{00}^T\mathbf{B}_{01} + \mathbf{A}_{10}^T\mathbf{B}_{11}) z^5 \\ &\quad + (*) z^6 + (\mathbf{A}_{01}^T\mathbf{B}_{01} + \mathbf{A}_{11}^T\mathbf{B}_{11}) z^7, \end{aligned}$$

where (*) in the expression refers to an interference term that is not of interest to us. $\mathbf{A}^T(z)\mathbf{B}(z)$ is a matrix polynomial of degree 7 and can therefore be interpolated as long as eight distinct evaluations are obtained. In general, the result of Yu et al. [27] shows that the threshold of their scheme is $\tau = pmn + p - 1$. The scheme can be decoded efficiently via polynomial interpolation. However, the numerical stability issue in this case is even more acute as the degree of the fitted polynomial is $pmn + p - 2$, i.e., much higher.

Recently, some contributions in the literature have attempted to address the numerical stability issues associated with polynomial-based approaches. In Das et al. [19], the authors demonstrate that convolutional codes can be used for matrix multiplication as well. They also demonstrate a computable upper bound on the worst-case condition number of the recovery matrices. This approach allows for schemes that are significantly better in terms of the numerical stability. Subramaniam et al. [22] consider choosing the encoding matrices (each entry i.i.d.) from a continuous distribution. Fahim and Cadambe [18] propose an alternative approach where the underlying polynomial scheme now operates on the basis of orthogonal polynomials, such as Chebyshev polynomials. They show that the condition number of the recovery matrices can be upper-bounded polynomially in the system parameters (as long as the number of stragglers is a constant), unlike real Vandermonde matrices, where the condition number grows exponentially. The recent work of Ramamoorthy and Tang [29] presents a different approach wherein polynomials are evaluated at structured matrices, such as circulant permutation and rotation matrices. The worst-case condition numbers obtained by this scheme are much lower than those obtained by Fahim and Cadambe [18].

Example 6

We now present an experimental comparison of different approaches for computing $\mathbf{A}^T\mathbf{B}$ with $r = w = 9{,}000$ and different t (see Table 2). We set up a cluster in the Amazon Web

Table 2. A comparison of the different schemes in terms of average worker computation time, total communication time (in parentheses), average decoding time, and worst-case condition number.

	Worker Computation and Communication Time (s)				
Methods	$t = 12{,}000$	$t = 18{,}000$	$t = 24{,}000$	**Decoding Time (s)**	**Condition Number**
Polynomial Codes* [17]	6.8(5)	10.8(6.9)	14.2(8.6)	1.24	24,753.93
Orthogonal Polynomial Codes* [18]	6.8(5)	10.8(6.9)	14.2(8.6)	1.24	266.59
Random Khatri–Rao Product Codes* [22]	6.8(5)	10.8(6.9)	14.2(8.6)	0.2	128.95
All-Ones Convolutional Codes† [19]	7.9(6.7)	12.2(8.9)	15.7(11.9)	0.24	95.198
Random Convolutional Codes† [19]	7.9(6.7)	12.2(8.9)	15.7(11.9)	0.41	87.093

*$\gamma_A = \gamma_B = 1/3$.
†$\gamma_A = \gamma_B = 2/5$.

Services cloud with one `t2.2xlarge` machine as the master node and $N = 11$ `t2.small` worker nodes. We considered a system with $p = 1, m = n = 3$ so that the threshold $\tau = 9$. The entries in Table 2 correspond to the worst-case computation time of each worker node for different values of t. We picked the set of workers that correspond to the worst condition number for the different schemes. For these methods, the worker computation time increases roughly linearly with t, while the decoding time does not change. The computational load on the worker nodes in the convolutional code approach is slightly higher than that in the polynomial code approach. This difference can be made as small as desired with higher subpacketization [19]. Note, however, that the condition number of the convolutional code is multiple orders of magnitude smaller. Our code implements the to and from communication from the master node to the workers sequentially; parallel implementations can further reduce the communication time.

Opportunities for future work

We hope the discussion in the preceding sections has convinced the reader that the area of coded matrix computation is a growing one and that there is ample scope to contribute toward it in various ways. MATLAB and Python code for several of the schemes in this survey article can be downloaded from https://github.com/anindyabijoydas/StragglerMitigateConvCodes [30]. We now outline some outstanding issues that require closer attention from the research community as a whole.

The vast majority of work in this area has considered distributed schemes for computing $\mathbf{A}^T\mathbf{B}$ for arbitrary matrices $\mathbf{A}$ and $\mathbf{B}$. However, in several practical scenarios, these matrices are sparse. This can change the computational complexity calculation significantly. We illustrate this by considering matrix–vector multiplication. If $\mathbf{A}$ (of dimension $t \times r$) is such that each column contains at most s nonzero entries, then computing $\mathbf{A}^T\mathbf{x}$ takes $\approx 2rs$ flops. Suppose that we apply the polynomial solution of the section "Distributed Vector–Matrix Multiplication." In this situation, each coded matrix $\tilde{\mathbf{A}}_l$ has approximately sm nonzero entries per column in the worst case (assuming $sm < t$). The worker node that computes $\tilde{\mathbf{A}}_l^T\mathbf{x}$ will therefore require $(1/m) \times 2rsm = 2rs$ flops. This means that in the worst case each worker node has the "same" computational load as computing $\mathbf{A}^T\mathbf{x}$, i.e., the computational advantage of distributing the computation may be lost. Table 3 tabulates the time for computing $\tilde{\mathbf{A}}_l^T\mathbf{x}$ (for a $30{,}000 \times 30{,}000$ $\mathbf{A}$) using the solution of Lee et al. [20] for a system with $N = 15$ worker nodes with a threshold of $\tau = m = 12$ for two kinds of sparse matrices:

1) an $\mathbf{A}$ that has the β-diagonal structure where only the diagonal and β off-diagonal terms are nonzero
2) an $\mathbf{A}$ where the nonzero entries are chosen at random.

Table 3 also lists the time of computing an uncoded matrix vector product, i.e., $\mathbf{A}_i^T\mathbf{x}$. It is clear that the worker node computation time increases significantly for the coded case. This is an issue with other papers [16], [17], [19], [24], [27] as well. The fountain-coding approach for the matrix–vector case [25] fares better here because with high probability the linear combination generated by the master node has low weight. However, Mallick et al. [25] do not provide provable guarantees on the recovery threshold and do require rather high values of m. This was also considered in Wang et al. [31] for the matrix–matrix case, though it is unclear whether their scheme respects the storage constraints on the workers as formulated in the section "Problem Formulation." The recent work of Wang et al. [32] makes progress on this problem. Wang et al. [32] define the *computational load* of a given coding solution in the matrix–vector case as the number of nonzero elements of the corresponding coding matrix. Their paper contains a discussion about lower bounds and achievability schemes for this metric.

Throughout this review article, we have highlighted the role of embedding an erasure code into a distributed matrix computation problem. We have shown that in the computation context, special attention needs to be paid to the numerical stability of the recovery of $\mathbf{A}^T\mathbf{B}$. Much existing work does not provide guarantees on the worst-case or average-case condition numbers. This is an important direction that needs to be pursued. There have been some initial results in this area [18], [19], [22], [29], but much remains to be done.

Table 3. A comparison of worker computation times when A is sparse.

Percentage of Zero Entries	**90%‡**	**80%‡**	**70%‡**
Time for uncoded case (ms)*	11.9(13.3)	22.1(22.7)	36.8(35.4)
Time for coded case (ms)†	109(83.8)	110.1(104.3)	122.2(108.2)

*The worker time for finding $\mathbf{A}_i^T\mathbf{x}$.
†The time to find $\tilde{\mathbf{A}}_i^T\mathbf{x}$.
‡The number in parentheses is for the case when $\mathbf{A}$ has a β-diagonal structure and the other number is for a sparse random $\mathbf{A}$.

The majority of existing work only deals with the recovery threshold (see the section "Problem Formulation"), which is in one-to-one correspondence with treating an erasure as a failed node. However, recovery threshold(II) considers a more fine-grained model, where different worker nodes operate at different speeds. The systematic design of schemes that provably leverage partial computations by the worker nodes is interesting. Ramamoorthy et al. [16] consider the case of matrix–vector multiplication, but systematic extensions to the matrix–matrix multiplication case would be of interest.

Authors

Aditya Ramamoorthy (adityar@iastate.edu) received his B.Tech. degree in electrical engineering from the Indian Institute of Technology, Delhi, in 1999 and his M.S. and Ph.D. degrees from the University of California, Los Angeles, in 2002 and 2005, respectively. He served as an editor of *IEEE Transactions on Information Theory* from 2016 to 2019 and *IEEE Transactions on Communications* from 2011 to 2015. He is the recipient of the 2012 Iowa State University's Early Career Engineering Faculty Research Award, the 2012 National Science Foundation CAREER Award, and the Harpole-Pentair professorship in 2009 and 2010. His research focus is in the areas of information theory and coding theory and their applications in a variety of domains such as distributed computation, machine learning and caching. He is a Senior Member of the IEEE.

Anindya Bijoy Das (abd149@iastate.edu) received his B. Sc. degree in electrical and electronic engineering from Bangladesh University of Engineering and Technology, Dhaka, in 2014, and his M.Eng. degree in electrical engineering from Iowa State University, Ames, in 2018. He is currently working toward his Ph.D. degree in the Department of Electrical and Computer Engineering at Iowa State University. His research interests include coding theory and machine learning.

Li Tang (litang@iastate.edu) received his B.E. degree in mechanical engineering and his M.S. degree in electrical and information engineering from Beihang University, Beijing, China, in 2011 and 2014, respectively. He is currently working toward his Ph.D. degree in the Department of Electrical and Computer Engineering at Iowa State University, Ames IA. His research interests include network coding and channel coding.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, and, and Y. Bengio, *Deep Learning*. Cambridge: MIT Press, 2016.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Conf. Networked Systems Design and Implementation (NSDI)*, 2013, pp. 185–198.

[3] N. B. Shah, K. Lee, and K. Ramchandran, When do redundant requests reduce latency? 2013. [Online] Available: https://arxiv.org/abs/1311.2851

[4] G. Joshi, E. Soljanin, and G. Wornell, "Efficient redundancy techniques for latency reduction in cloud systems," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 2, pp. 12:1–12:30, 2017. doi: 10.1145/3055281.

[5] B. Li, A. Ramamoorthy, and R. Srikant, "Mean-field analysis of coding versus replication in large data storage systems," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 1, pp. 3:1–3:28, 2018. doi: 10.1145/3159172.

[6] D. Wang, G. Joshi, and G. W. Wornell, "Efficient straggler replication in large-scale parallel computing," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 4, no. 2, pp. 7:1–7:23, Apr. 2019. doi: 10.1145/3310336.

[7] C. M. Bishop, *Pattern Recognition and Machine Learning*. Berlin: Springer-Verlag, 2006.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1097–1105. doi: 10.1145/3065386.

[9] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2016, pp. 2100–2108.

[10] R. A. Horn and C. R. Johnson, *Matrix Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1990.

[11] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 2004.

[12] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 100, no. 6, pp. 518–528, 1984. doi: 10.1109/TC.1984.1676475.

[13] J.-Y. Jou and J. A. Abraham, "Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures," *Proc. IEEE*, vol. 74, no. 5, pp. 732–741, 1986. doi: 10.1109/PROC.1986.13535.

[14] G. Sobczyk, "Generalized Vandermonde determinants and applications," *Aportaciones Matematicas, Serie Comunicaciones*, vol. 30, pp. 203–213, 2002.

[15] V. Y. Pan, "TR-2013003: Polynomial evaluation and interpolation—Fast and stable approximate solution," City Univ. of New York, 2013. [Online] Available: https://academicworks.cuny.edu/gc_cs_tr/378/

[16] A. Ramamoorthy, L. Tang, and P. O. Vontobel, "Universally decodable matrices for distributed matrix-vector multiplication," in *Proc. IEEE Int. Symp. Information Theory (ISIT)*, 2019, pp. 1777–1781. doi: 10.1109/ISIT.2019.8849451.

[17] Q. Yu, M. Maddah-Ali, and S. Avestimehr, "Polynomial codes: An optimal design for high-dimensional coded matrix multiplication," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 4403–4413.

[18] M. Fahim and V. R. Cadambe, "Numerically stable polynomially coded computing," in *Proc. IEEE Int. Symp. Information Theory (ISIT)*, 2019, pp. 3017–3021. doi: 10.1109/ISIT.2019.8849468.

[19] A. B. Das, A. Ramamoorthy, and N. Vaswani, Random convolutional coding for robust and straggler resilient distributed matrix computation. 2019. [Online] Available: https://arxiv.org/abs/1907.08064

[20] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, 2018. doi: 10.1109/TIT.2017.2736066.

[21] V. Pan, "How bad are Vandermonde matrices?" *SIAM J. Matrix Anal. Appl.*, vol. 37, no. 2, pp. 676–694, 2016. doi: 10.1137/15M1030170.

[22] A. M. Subramaniam, A. Heidarzadeh, and K. R. Narayanan, "Random Khatri-Rao-product codes for numerically-stable 500 distributed matrix multiplication," in *Proc. Allerton Conf. Communication, Control, and Computing (Allerton)*, Sept. 2019, pp. 253–259. doi: 10.1109/ALLERTON.2019.8919859.

[23] W. P. Wardlaw, "Matrix representation of finite fields," *Math. Mag.*, vol. 67, no. 4, pp. 289–293, 1994. doi: 10.2307/2690850.

[24] A. B. Das and A. Ramamoorthy, "Distributed matrix-vector multiplication: A convolutional coding approach," in *Proc. IEEE Int. Symp. Information Theory (ISIT)*, 2019. doi: 10.1109/ISIT.2019.8849395.

[25] A. Mallick, M. Chaudhari, and G. Joshi, "Fast and efficient distributed matrix-vector multiplication using rateless fountain codes," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 8192–8196. doi: 10.1109/ICASSP.2019.8682347.

[26] A. E. Yagle, "Fast algorithms for matrix multiplication using pseudo-number-theoretic transforms," *IEEE Trans. Signal Process.*, vol. 43, no. 1, pp. 71–76, 1995. doi: 10.1109/78.365287.

[27] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," in *Proc. IEEE Int. Symp. Information Theory (ISIT)*, 2018, pp. 2022–2026. doi: 10.1109/ISIT.2018.8437563.

[28] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *IEEE Trans. Inf. Theory*, vol. 66, no. 1, pp. 278–301, 2019. doi: 10.1109/TIT.2019.2929328.

[29] A. Ramamoorthy and L. Tang, Numerically stable coded matrix computations via circulant and rotation matrix embeddings. 2019. [Online] Available at: https://arxiv.org/abs/1910.06515

[30] "Straggler mitigation codes," GitHub, San Francisco. Accessed on: Mar. 2, 2020. [Online] https://github.com/anindyabijoydas/StragglerMitigateConvCodes

[31] S. Wang, J. Liu, and N. Shroff, "Coded sparse matrix multiplication," in *Proc. Int. Conf. Machine Learning (ICML)*, 2018, pp. 5152–5160.

[32] S. Wang, J. Liu, N. Shroff, and P. Yang, "Computation efficient coded linear transform," in *Proc. Int. Conf. Artificial Intelligence and Statistics*, 2019.

SP