INVITED PAPER

Addressing Unreliability in Emerging Devices and Non-von Neumann Architectures Using Coded Computing

By Sanghamitra Dutta, *Graduate Student Member, IEEE*, Haewon Jeong, *Member, IEEE*, Yaoqing Yang, *Member, IEEE*, Viveck Cadambe, Tze Meng Low, and Pulkit Grover, *Senior Member, IEEE*

ABSTRACT | Computing systems are evolving rapidly. At the device level, emerging devices are beginning to compete with traditional CMOS systems. At the architecture level, novel architectures are successfully avoiding the communication bottleneck that is a central feature, and a central limitation, of the von Neumann architecture. Furthermore, such systems are increasingly plagued by unreliability. This unreliability arises at device or gate-level in emerging devices, and can percolate up to processor or system-level if left unchecked. The goal of this article is to survey recent advances in reliable computing using unreliable elements, with an eye on nonsilicon and non-von Neumann architectures. We first observe that instead of aiming for generic computing problems, the community could use "dwarfs of modern computing," first noted in the high-performance computing (HPC) community, as a starting point. These computing problems are the basic building blocks of almost all scientific computing, machine learning, and data analytics today. Next, we survey the state of the art in "coded computing," which is an emerging area that advances on classical algorithm-based fault-tolerance (ABFT) and brings a fundamental information-theoretic perspective. By weaving error-correcting codes into a computing algorithm, coded computing provides dramatic improvements on solutions, as well as obtains novel fundamental limits, for problems that have been open for more than 30 years. We introduce existing and novel coded computing techniques in the context of "coded dwarfs," where a specific dwarf's computation is made resilient by applying coding. We discuss how, for the same redundancy, "coded dwarfs" are significantly more resilient compared to classical techniques such as replication. Furthermore, by examining a widely popular computation task—training large neural networks—we demonstrate how coded dwarfs can be applied to address this fundamentally nonlinear problem. Finally, we discuss practical challenges and future directions in implementing coded computing techniques on emerging and existing nonsilicon and/or non-von Neumann architectures.

KEYWORDS | Computer errors; distributed algorithms; distributed computing; distributed processing; error correction; error correction codes; fault tolerance; fault tolerant systems; high performance computing (HPC); large-scale systems; parallel architectures; parallel machines; parallel processing; supercomputers.

Manuscript received February 1, 2019; revised March 19, 2020; accepted March 20, 2020. This work was supported by NSF under Grant 1763561. *(Sanghamitra Dutta, Haewon Jeong, and Yaoqing Yang contributed equally to this work.) (Corresponding author: Pulkit Grover.)*

Sanghamitra Dutta, **Haewon Jeong**, **Tze Meng Low**, and **Pulkit Grover** are with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: pulkit@cmu.edu).

Yaoqing Yang is with the Department of Electrical Engineering and Computer Science, University of California at Berkeley (UC Berkeley), Berkeley, CA 94720 USA.

Viveck Cadambe is with the Department of Electrical Engineering, Penn State University, State College, PA 16801 USA.

Digital Object Identifier 10.1109/JPROC.2020.2986362

I. INTRODUCTION

With the imminent saturation of Moore's law and Dennard scaling and the ever-increasing rate of incoming data, the current scaling of transistors cannot keep up with the data growth (also called "data deluge gap") [1]. This has resulted in the community adopting novel approaches at all levels: from circuits and devices, all the way to system-level innovations. At device level, departing from the well-established CMOS technologies, designers are exploring novel nonsilicon devices ranging from spintronics, graphene, mechanical switches, to analog computation engines. Alongside, to improve performance, at chip and system-level, the community has been seeking solutions through massively parallel and distributed systems that depart radically from the conventional von Neumann architectures. For instance, nowadays, deep neural networks (DNNs) are trained over hundreds of graphics processing unit (GPU)-accelerated servers, where each GPU has thousands of cores [2]. The communication bottleneck of accessing data in von Neumann architectures is, to some extent, alleviated in massively distributed computing [3, p.3] by bringing compute to the data and pinning the data at distributed computers for reuse (e.g., in iterative machine learning and data analytics [4]). This simple departure from von Neumann architectures shifts the major bottleneck from the memory–compute interface to the message-passing interface between distributed machines, and the difficulty shifts from conducting some reads and writes to keeping thousands of machines and their communication links simultaneously reliable.[1]

All of these approaches suffer from different forms of unreliability. In emerging devices, unreliability can be imposed by low energy/low volume requirements, for example, due to the use of analog systems aimed at performing tasks implemented digitally today. The resulting errors, if left unchecked, accumulate and propagate, an effect known as "information dissipation" [5], [6]. The scale of parallelism in upcoming systems brings about new reliability challenges as well [7]–[10]. To see how scale affects reliability, consider the Fugaku supercomputer which is planned to be fully operational in 2021. The Fugaku system consists of 150 000 physical nodes with a total of 8 million cores [11]. To achieve a system-level mean-time-between-failures (MTBFs) of 24–48 h, the MTBF of each node must be 411–822 years. Such nodes are difficult to design, implement, and test, and leave little room for unexpected reliability issues (e.g., dirty power and unexpected early wear-out [12], [13]). Another type of unreliability, that is more common in shared cloud computing environments with thousands of distributed nodes, is "straggling." Stragglers are nodes that take substantially longer time to complete their computation task than their counterparts, becoming system-level bottlenecks. Straggling can occur for a multitude of reasons [14] including hardware heterogeneity [15]–[17], hardware failures [18], [19], and various OS-related effects [20], [21]. It was observed that the slowest nodes can be $>8\times$ slower than the median, and addressing straggling can reduce the average completion time by 47% [19].

An ambitious example of how emerging devices depart from both classical CMOS and classical von Neumann architecture are novel analog computation engines. These engines perform small analog computations efficiently, and can be used as building blocks in a larger system. How should a designer choose which computation these engines implement? The choice must be driven by applications. Indeed, the first such engines [22]–[26] implemented dot products, which are building blocks of dense linear algebraic operations frequently used in machine learning and scientific computing (see Section III). What can be other such choices? In the mid-2000s, the high-performance computing (HPC) community arrived at several such canonical computations that form the basic building blocks of modern-day computing, including machine learning, data analytics, and scientific computing. We propose that small computations within these dwarfs can serve as goals for analog computation engine designers to design the next class of systems. Initially, seven so-called "dwarfs of computation" [27] were identified (a set which was later expanded [28]), each of which is a class of computations that share similar computation and communication patterns.

The challenge of interest in this article is one of integration: how do we reliably implement the computation dwarfs by integrating unreliable components? To address this question, we leverage techniques developed recently in the information theory community, collectively called "coded computing." These techniques incorporate error-correcting codes (ECCs) into computation, advancing (sometimes dramatically) on classical techniques in the field of algorithm-based fault tolerance (ABFT) [29]–[31]. Coded computing has generated exciting results in recent years [32]–[34], including notable breakthroughs such as obtaining storage-optimal solutions to error-resilient matrix multiplication [35]–[37], obtaining the first solution to linear transforms with all elements being error-prone [38], and addressing the fundamental question of error-resilient neural network training [39], [40] (inspired from von Neumann's 60-year-old work on computing reliably using unreliable organisms[2]). In the spirit of dwarfs of computation, we call the coded computing solutions for computing dwarfs, "coded dwarfs." Compared to traditional techniques, such as replication or checkpointing, coded dwarfs can obtain a target error tolerance with substantially smaller overhead. For example, for distributed matrix multiplication, replication-based schemes require

[1]Note that popular instantiations of non-von Neumann architectures, such as single-instruction–multiple-data (SIMD) and cellular automatons, share the same fundamental bottleneck with parallel and distributed computing.

[2]In his seminal work [41], von Neumann also referred to the McCulloch–Pitts model of a neuron [42], an extremely simple example of an artificial neuron, thus provoking a question that is very relevant in today's world: how to train reliably using unreliable computational units?

two times redundancy for detecting an error and three times redundancy [also known as triple modular redundancy (TMR)] for correcting an error. On the other hand, coded matrix multiplication can provide a single error detection and correction capability with a small (asymptotically negligible) redundancy.

To make this integration issue more concrete, consider the following example. As we noted, researchers in circuits and devices have implemented specialized analog engines for computing dot products of two vectors (see [23] and [24] among others). How do we integrate these engines to perform modern computations, for example, for training DNNs (see Section VII)? Being emerging technologies, their yield rates can be small, and failure rates in run-time high. Furthermore, for some inputs, the errors are small, while for others, they can be quite large. Despite being carefully crafted using state-of-the-art technologies, these systems can suffer from significant errors that are in some cases dependent on input vectors (see [23]). Improved fabrication techniques can reduce these issues, but not eliminate them. Thus, the system needs to be engineered in a way that can address these errors before they snowball into large errors affecting the eventual decision or inference.

The goal of this article is to survey recent advances in error-correction techniques for computing that can help address unreliability in the future computational systems that arises due to the use of emerging devices. Furthermore, this article also discusses techniques to address errors/faults that percolate to the level of a single processor or node, for example, in massively distributed non-von Neumann computing systems. At that level, errors, faults, straggling, network delays, etc., exist in today's systems, and are increasing as these systems become more complex and more constrained. As novel technologies start forming the building blocks of these systems, the unreliability will only increase. Understanding and advancing techniques for addressing different types of unreliabilities is thus an urgent and important issue.

In the remainder of this article, we introduce coded dwarfs, that is, computation dwarfs that are made resilient to errors using coded computing techniques. In Sections III–VI, we review and summarize coded computing techniques applied to four dwarfs: dense linear algebra, sparse linear algebra, spectral methods, and MapReduce. In Section VII, we discuss how coded dwarfs can be utilized for important machine learning applications through an example of training a neural network. Finally, in Section VIII, we identify some important open problems, and lay out our vision on how information theorists, systems experts, and circuit designers can work together to build resilience in the next generation computing systems.

II. BACKGROUND AND NOTATION

A. Computation Dwarfs

What are the indispensable computation building blocks in modern-day computing? Colella [27] introduced the idea of seven dwarfs of computation in his 2004 presentation, which are seven building blocks[3] for modern day scientific computing: 1) dense linear algebra; 2) sparse linear algebra; 3) spectral methods; 4) N-body methods; 5) structured grids; 6) unstructured grids; and 7) Monte Carlo (MapReduce). The seven dwarfs have served as a guideline for building and testing a parallel system [43], [44]. In this article, we focus on four of the seven dwarfs for which coding strategies have studied: dense and sparse linear algebra, spectral methods, and MapReduce.

B. ECCs in Communication and Storage

ECCs are commonly used in communication and storage systems with the objective of recovering lost or corrupted data through adding redundancy. We now introduce a few mathematical notation required to describe ECCs. We will use uppercase bold fonts for matrices (e.g., $\mathbf{A}$) and lowercase bold fonts for vectors (e.g., $\mathbf{x}$). Let $\mathbf{a}$ denote a length-m input vector. By adding $P-m$ redundant symbols, we want to encode this input vector into a length-P coded vector $\mathbf{a}_{\text{coded}}$. A popular choice of encoding function is a linear mapping which can be represented as

$$\mathbf{a}_{\text{coded}} = \mathbf{G}^T\mathbf{a} \tag{1}$$

where $\mathbf{G}$ is a m-by-P matrix, called a generator matrix. An important question is: how many lost symbols can we recover if we add $P-m$ redundant symbols? One might hope to tolerate $P-m$ erasures since $P-m$ redundant symbols were added. Indeed, this is provably the best performance any encoding function can achieve, and there exists a linear encoding scheme that achieves this. Coding schemes that achieve this are called maximum distance separable (MDS) codes. Another important class of codes is systematic codes. In systematic codes, the first m symbol of $\mathbf{a}_{\text{coded}}$ would be just a copy of the original message $\mathbf{a}$, and the last $P-m$ symbols would be linear combinations of $\mathbf{a}$. Encoded symbols in systematic codes are often called parity symbols or checksums.

C. Non-von Neumann Computing Systems and Unreliabilities

In most parts of this article, we consider distributed/parallel computing systems where each compute node is equipped with its own memory. In distributed systems, principal unreliability concerns are node failures (nodes that crash in the middle of computation) and stragglers (nodes that respond substantially slower than others). For both, we will consider them as "erasures" as one can discard results from failed or straggling nodes. When designing a coding strategy for these systems, some important overheads to consider are communication, computation

[3]The set was later expanded to 13 dwarfs by Asanovic *et al.* [28] to include graphical models, finite state machines, among others.

per node, memory usage per node, and the required number of distributed nodes.

Another non-von Neumann system that we consider is low-energy compute engine for specific computations. Developing specialized energy-efficient accelerators has been a crucial part of the roadmap to nonsilicon, non-von Neumann computing. However, due to low energy constraints and the use of emerging materials, gates on these circuits can have high unreliability. We will consider gate-level transient errors (see Section III-C), as well as large analog errors in analog compute engines (see Section III-A2), in these emerging devices and systems.

D. Performance Metrics

In distributed systems, we use P to denote the total number of nodes. We define the recovery threshold (K) as the number of nodes that have to compute successfully out of a total of P nodes, to be able to obtain the entire final result. Throughout this article, recovery threshold is often used as a performance metric of a coding strategy. Lower recovery threshold means that we can tolerate more faults and errors with the same number of nodes. Depending on the problem, we also examine some other performance metrics, for example, expected runtime in the presence of straggling nodes, convergence rate in the case of iterative algorithms, etc.

III. CODED DWARF 1: DENSE LINEAR ALGEBRA

The first dwarf, dense linear algebra comprises a large set of operations on dense vectors and matrices, with abundant applications in scientific computing and machine learning [45]. The operations in dense linear algebra are largely classified into three categories: vector–vector operations (BLAS Level 1), matrix–vector operations (BLAS Level 2), and matrix–matrix operations (BLAS Level 3). As these operations are essential in scientific computing and machine learning [45], some of the first analog computation engines have been built to support them [23], [24]. At system-level, there exist multiple libraries implementing these operations (e.g., BLAS and LAPACK).

In this section, we review coded computing techniques for fault/error resilient matrix–vector ($\mathbf{Ax}$) or matrix–matrix ($\mathbf{AX}$) multiplication. These linear transforms are the building blocks of various machine learning problems [45], (e.g., regression, and classification) and are also used in acquiring and preprocessing the data through Fourier transforms, wavelet transforms, filtering, etc. In Sections III-A and III-B, we review results on combating stragglers/node failures on distributed systems. In Section III-C, we turn our attention to emerging devices for binary linear transforms and introduce a strategy called ENcoded COmputation with Decoders EmbeddeD (ENCODED) [38] that studies how to make the circuit resilient when all gates are prone to errors.

A. Coded Dense Matrix—Vector Products

Can we perform reliable matrix–vector products ($\mathbf{Ax}$) using unreliable hardware, for example, distributed system of processing nodes prone to faults and stragglers, or emerging devices such as analog dot-product engines?

1) MDS Codes: A recent work [32] proposed the use of MDS codes [46] for coded matrix–vector products, which can be viewed as a rediscovery of the ABFT approach adopted in the original work of Huang and Abraham [29]. Consider the problem of multiplying a matrix $\mathbf{A}$ with a vector $\mathbf{x}$ such that the matrix $\mathbf{A}$ is too large to be stored on any single node. We start with a simple example.

a) Simple example: Assume that each node can only store half of the matrix $\mathbf{A}$. Divide $\mathbf{A}$ horizontally as follows: $\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix}$. Let two nodes store these two submatrices in their local memories and compute $\mathbf{A}_1\mathbf{x}$ and $\mathbf{A}_2\mathbf{x}$, respectively. The master node waits for both the nodes to finish their computation, and then produces the final result. Note that the two nodes can compute these matrix–vector products using different implementations.

In this setting, if any one node is prone to faults or stragglers, then the entire computation is affected. One method to ensure error tolerance is to use two more nodes and replicate the computational tasks. The master node can now tolerate one faulty node (essentially an erasure) in the worst case and requires any three out of the four nodes to finish the computation. Although it can sometimes tolerate two erasures, it fails to reconstruct the result if the two faulty nodes are the nodes performing replicas of the same computation, for example, the two nodes computing $\mathbf{A}_1\mathbf{X}$. In this case, the recovery threshold using replication is $K = 3$ because the master node can recover the result using the partial computational results from any three out of the four nodes.

The use of MDS codes reduces the recovery threshold further as compared to the replication strategy. In this case, we choose four linear combinations of $\mathbf{A}_1$ and $\mathbf{A}_2$ as follows: $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_1 + \mathbf{A}_2$, and $\mathbf{A}_1 - \mathbf{A}_2$, respectively, and store one at each of the four nodes. Each node multiplies its stored encoded submatrix with $\mathbf{x}$ and sends the result to the master node. Interestingly, the master node now only requires any two out of the four computations $\mathbf{A}_1\mathbf{x}$, $\mathbf{A}_2\mathbf{x}$, $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{x}$, and $(\mathbf{A}_1 - \mathbf{A}_2)\mathbf{x}$ to be able to successfully reconstruct $\mathbf{A}_1\mathbf{x}$ and $\mathbf{A}_2\mathbf{x}$. The recovery threshold is thus reduced to 2 from 3. This provides better fault resilience while using the same number of processing nodes, that is, four and the same memory and computational resources at each node as the replication strategy. It can be generalized to using a (P, m) MDS code: the matrix $\mathbf{A}$ is split in to m horizontal blocks and the MDS code generates P linear combinations of the m submatrices such that any m out of those P encoded blocks can reconstruct the final output $\mathbf{Ax}$.

b) Performance: Lee *et al.* [32] demonstrated improvements in performance using MDS coded computing on straggling-prone nodes (through both theoretical

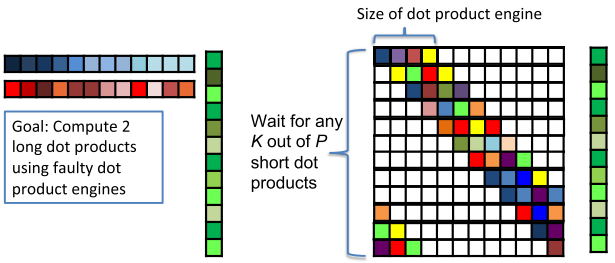

Fig. 1. ***Short-dot codes generate a larger set of short redundant dot products such that a subset of them is sufficient.***

analyses and experimental results) over replication and uncoded strategies. While it has not been tested experimentally yet, the techniques naturally extend to tolerate errors (see [47] for how many errors can be tolerated), and we believe that it will be as effective in systems with error-prone nodes.

2) Short-Dot Codes: Short and fat linear transforms on high-dimensional data arise frequently in dimensionality reduction techniques such as principal component analysis (PCA), linear discriminant analysis (LDA), and random projections. Short-dot codes [34] provide a novel strategy of performing fault-tolerant linear transforms where, instead of computing long dot products as required in the original linear transform, it constructs a larger number of redundant and short dot products that can be computed faster and more efficiently at individual processing nodes. For instance, one can connect emerging devices such as analog dot-product engines that can only compute short dot products [22]–[24] in order to compute a large matrix–vector product.

More specifically, short-dot codes [34] address the problem of distributed matrix–vector product (e.g., $\mathbf{Ax}$) using P faulty processing nodes under the constraint that each node can only access s elements of the long vector $\mathbf{x}$ of length N. The key novelty in this article is that redundant and short dot products computed on different nodes can be synthesized in an error-resilient fashion to recover the matrix–vector product as illustrated in Fig. 1. In Fig. 1, our goal is to compute two long dot products of length $N(=12)$ using $P(=12)$ small and faulty dot product engines that can only compute dot products of length $s(=4)$. Applying a $(P, 2)$ MDS code naively to this problem would result in P dot products of length N each and not reduce the length of the individual dot products. On the other hand, the short-dot codes enable us to generate a redundant set of P vectors from the original two vectors such that: 1) each vector only has $s(=4)$ nonzero elements and 2) any K out of these P vectors can linearly span the original two vectors, that is, we can recover the two original dot products from any K out of these P dot products. An application of these codes to resilient data-parallel gradient descent [48]–[51] is discussed in Section VII.

a) Performance: Dutta *et al.* [34] compared the expected computation time of short-dot codes with competing strategies. Fig. 2 shows that short-dot codes tradeoff between the length of the dot products computed at each node (s) and the recovery threshold (K). The MDS coding strategy and the uncoded strategy are two special cases of short-dot. Short-dot codes can tolerate $P-K$ erasures when the decoder knows which outputs are faulty, for example, for the straggler problem it can identify which nodes did not finish. It is also shown in [34] that short-dot codes can tolerate $\lfloor((P-K)/2)\rfloor$ errors when the decoder does not know which outputs are erroneous and has to correct as well as detect errors.

b) Optimality of short-dot codes: Suh *et al.* [52] obtain a fundamental limit (that improves on the fundamental limit in [34]) to demonstrate that short-dot codes are optimal in their tradeoff between the length of the dot product (s) and recovery threshold (K).

In another related work, Beckman *et al.* [22] explored the intriguing problem of error-resilient integer matrix–vector products $\mathbf{Ax}$ using dot product engines where the matrix $\mathbf{A}$ is realized as a crossbar array. Conductors for each row and column form a grid and programmable nanoscale resistors (e.g., memristors) at the junctions of the grid have the conductance proportional to the corresponding element of $\mathbf{A}$. The goal in this article [22] is not to make individual dot products short but to address computational errors arising from a variety of factors such as inaccuracies while programming the resistors in the crossbar, noise while reading the currents, junctions in the crossbar becoming shorted, etc. Toward this goal, they proposed using systematic ECCs and analyzed the error tolerance in terms of L_1 metric and Hamming metric.

B. Coded Matrix–Matrix Multiplication

In this section, we consider the problem of multiplying two matrices $\mathbf{A}$ and $\mathbf{X}$ using a set of P worker nodes such that each node can only store a fixed $1/m$ fraction of each

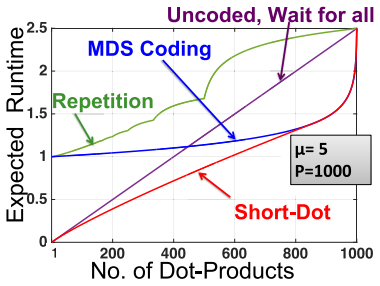

Fig. 2. ***Expected computation time plotted against total number of dot products to be computed: Short-dot is faster than MDS when the total number of dot products is much less than P, and outperforms uncoded when the total number of dot products approaches P. To model straggling, the runtime of each node is assumed to be distributed as $s + Exp(\mu/s)$ for computing a dot-product of length s [34, Fig. 5].***

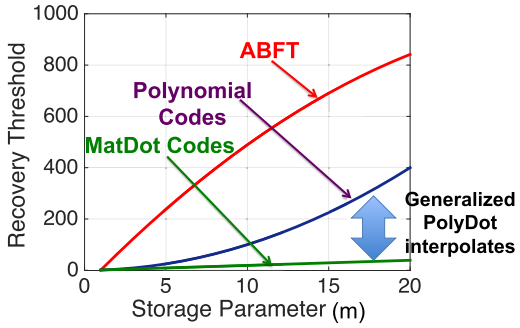

Fig. 3. ***Scaling of recovery threshold with storage parameter m, i.e., when each node can store a fraction 1/m of each of the matrices being multiplied. Total number of nodes is P = 1000. MatDot codes achieve the lowest recovery threshold for the storage constrained matrix multiplication problem. Generalized PolyDot codes interpolate between MatDot codes and polynomial codes (figure from [47]).***

matrix. For simplicity, we assume that both the matrices are $N \times N$ square matrices. We measure performance in terms of erasure recovery threshold (for failures and stragglers). The results carry over to error recovery threshold using approaches similar to those for matrix–vector products. We will first introduce ABFT/Product codes which can be considered as an extension of MDS codes for matrix–vector multiplication. Then, we will discuss three code constructions and bring out the interesting tradeoff between communication costs and recovery threshold. The performance of the four strategies we discuss here is summarized in Fig. 3.

1) ABFT/Product Codes: ABFT techniques for matrix multiplication [29] as well as the recently proposed product codes [53] for this problem have a similar encoding technique. The first matrix **A** is split horizontally and the second matrix **X** is split vertically into m submatrices each. Next, the submatrices of **A** are encoded using a $(\sqrt{P}, m)$ MDS code to generate $\sqrt{P}$ submatrices such that any m out of them are sufficient to generate **A**. Similarly, the m submatrices of **X** are also encoded using another $(\sqrt{P}, m)$ MDS code to generate $\sqrt{P}$ coded submatrices. Next, every coded submatrix of **A** is multiplied with every coded submatrix of **X** in a separate worker node.

a) Performance: This strategy has a worst case recovery threshold of $K = 2(m-1)\sqrt{P} - (m-1)^2 + 1 = \Theta(\sqrt{P})$, even though in the average-case fewer nodes suffice [53].

2) Polynomial Codes: The worst case recovery threshold for this problem was improved in scaling sense using another code construction called polynomial codes [54]. In this coding technique, the matrices **A** and **X** are again split horizontally and vertically into m submatrices as before. Next, we use two carefully chosen polynomials[4] for encoding the input matrices, one for each of **A** and **X**. The coefficients of each polynomial are the submatrices of **A** and **X**, respectively. Each node gets different encoded versions of **A** and **X**, which are the evaluations of the encoding polynomials at different points. Then, it performs matrix multiplication on those encoded matrices and sends the computational result to the decoder. The judicious choice of the two encoding polynomials enables the decoder to reconstruct the final result from a subset of the worker nodes using polynomial interpolation.

a) Performance: The recovery threshold of this strategy is $K = m^2$, a scaling sense improvement over ABFT/product codes. Also, polynomial codes achieve the optimal communication cost.

3) MatDot Codes: In [35], a novel encoding technique called MatDot codes was proposed that outperform polynomial codes, achieving an even lower recovery threshold of $2m - 1$, albeit a higher communication cost and higher per-node computational complexity. Contrary to the partitioning of polynomial codes, in MatDot codes, the first matrix **A** is divided vertically and the second matrix **X** is divided horizontally. Then, we encode them using thoughtfully designed polynomials specific to this partitioning.

a) Performance: The recovery threshold is $K = 2m - 1$. This is a scaling sense improvement over m^2 achieved by polynomial codes. It is the best-known recovery threshold, and one can also prove that it is optimal under assumptions noted in Section III-B5.

4) Generalized PolyDot Codes: There is a tradeoff between the recovery threshold and the communication/computational costs. Polynomial codes have a higher recovery threshold of m^2, but have a lower communication cost of $\mathcal{O}(N^2/m^2)$ per worker node, and also a lower computational cost of $\mathcal{O}(N^3/m^2)$. On the other hand, MatDot codes have a lower recovery threshold of $2m-1$, but have a higher communication cost of $\mathcal{O}(N^2)$ per worker node, and a computational cost of $\mathcal{O}(N^3/m)$. A family of codes called generalized PolyDot codes [35], [55] (independently discovered as entangled polynomial codes [37], both of these works followed the discovery of MatDot codes [35]) bridge the gap between polynomial codes and MatDot codes and provide intermediate communication costs and recovery thresholds.

For this construction, we consider a slightly more general problem statement where we want to multiply two matrices **A** and **X** such that each node can store a fixed $1/m$ fraction of **A** and $1/m'$ fraction of **X**. Choose integers r, s, t such that $rs = m$ and $st = m'$. First, we partition **A** into an $r \times s$ grid of equal-sized subblocks and **X** into an $s \times t$ grid. The PolyDot framework for matrix multiplication (originally proposed in [35] and later improved in [37] and [55]) introduces two multivariate polynomials for encoding **A** and **X**. Computation at each node and decoding through polynomial interpolation are essentially the same as polynomial codes and MatDot codes. One exciting application of this family of codes is training DNNs, which we discuss in Section VII.

[4]Using polynomials is a common technique in coding theory. Some of the most well-known codes, e.g., Reed–Solomon codes (on whose construction polynomial codes are based), are constructed from polynomials.

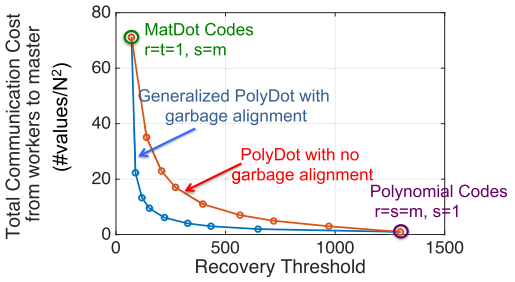

Fig. 4. ***Tradeoff between communication cost (from the workers to the master node) and recovery threshold of generalized PolyDot codes by varying r, s, and t for a fixed m = m′ = 36. MatDot codes have the lowest recovery threshold 2m − 1 = 71. The minimum communication cost is N², corresponding to polynomial codes, that have the largest recovery threshold m² = 1296. Generalized PolyDot codes bridge between these two strategies, improving the tradeoff using garbage alignment (figure from [47]).***

a) Performance: The recovery threshold of generalized PolyDot codes is $K = rst + s - 1$. It is achieved by a novel technique called "garbage alignment" that aligns some of the unwanted coefficients in the polynomials to reduce the degree of the final polynomial being interpolated. In Fig. 4, we illustrate the tradeoff between MatDot and polynomial codes, as achieved by generalized PolyDot codes.

5) Fundamental Limits and Other Works: The recovery threshold of MatDot codes improves in scaling sense over polynomial codes. Subsequent work [37] obtained a fundamental limit of $\min\{P, rs + st - 1\}$ for this problem when each input matrix is partitioned both vertically and horizontally under the assumption that the input encoding is linear and each node simply performs matrix multiplication on encoded matrices. These limits show that for the chosen partitioning ($r = t = 1, s = m$), MatDot codes are, in fact, optimal. The recovery threshold of MatDot codes also match in scaling sense with the more general, information-theoretic converse of $\max\{rs, st\}$ in [37]. However, for other kinds of partitioning, there is a gap between the fundamental limits and the best known achievability of $K = rst + s - 1$.

We also refer the reader to some recent works [56]–[66] in the coded computing community using interesting alternate techniques such as efficient task allocation, performance analysis, and utilizing partial task of slow workers. Recent work [67] has shown that although the MatDot code was originally proposed to reduce the storage overhead instead of communication cost, it naturally applies to the widely used communication-avoiding matrix multiplication algorithm [68] on a 3-D mesh, by exploiting the outer-product decomposition.

C. Coded Dense Linear Algebra Using Entirely Unreliable Components

In the discussion thus far, we have assumed that the computing engines are unreliable, but that the encoding/decoding mechanisms can be performed reliably. This is justified in many cases because the encoding/decoding complexity is much lower than that of the engine. In this section, we consider a more challenging problem, one where all elements are noisy. In fact, the study of this problem was initiated by von Neumann in 1956. In his model, the computation units and storage cells on a circuit are arranged in a fully distributed fashion, and different subsets of these units take turns to become active to carry out the computation. All the units, including the computation units and the storage cells, and the error correction mechanism, can have transient faults. Therefore, it is desired that the circuit structure itself, that is, the way that the computational units and the storage cells are wired together, is designed to be error-tolerant. Note that the main focus of this section is on the circuit implementations instead of the distributed computing models considered in Sections III-A and III-B, and also that we consider circuit-level transient faults instead of machine failures or stragglers.

Interestingly, on any single path of computational units, the computation error must accumulate, as discussed in the information dissipation work of Evans and Schulman [5, Lemma 2] (see also [6]). Thus, to reduce error accumulation, we should maximally distribute the information in the network, so that information can be integrated from different paths to fight the dissipation on each path. This leads to the idea of expander graphs, which have been used as building blocks of certain low-density parity-check (LDPC) codes [69]. Expander graphs are sparse graphs constructed such that each small subset of fewer than S vertices are connected to more than δS vertices for some positive constant δ. This strong connectivity property leads to reliable coding designs on the graph, ensuring that any wrong information contained in a small subset of vertices can be corrected using a local majority vote in a large neighborhood.

In [38], the first coding technique for computing binary linear transform using entirely unreliable components was proposed. The strategy is called "ENcoded COmputation with Decoders EmbeddeD," or "ENCODED" to highlight the critical aspect of embedded decoding units. At a high level, for computing $\mathbf{A} \cdot \mathbf{x}$ ($\mathbf{A}$ and $\mathbf{x}$ are binary), the computation is partitioned into multiple stages, so that errors at each stage do not accumulate and make the code unable to handle them. Then, low-complexity decoding is carried out in each stage, that is, we partition the decoding as well and interleave the computation stages with the "noisy" decoding stages (utilizing results from [70] to [71]) to maximally suppress errors with limited resources. Specifically, the matrix $\mathbf{A}$ is encoded into $\mathbf{A}_{\text{coded}}$, and the computation $\mathbf{A}_{\text{coded}} \cdot \mathbf{x}$ is partitioned into the summation of $x_i \cdot \mathbf{a}_i$, where x_i and $\mathbf{a}_i$ are, respectively, the ith entry of $\mathbf{x}$ and the ith column of $\mathbf{A}_{\text{coded}}$. This is to ensure that the iterative formula at each stage of the computation has the form $\mathbf{y}_i = \mathbf{y}_{i-1} + x_i \cdot \mathbf{a}_i$, so that the intermediate result at each stage is a codeword (in the noiseless case). It is shown, through both theoretical results and simulations, that the error

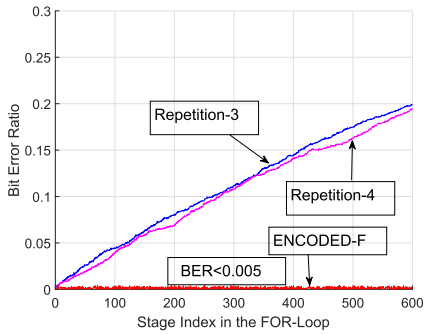

Fig. 5. ***ENCODED is able to keep errors contained even as the computation proceeds, while repetition/replication-type approaches cannot. Here, the x-axis represents the index of the interleaved compute-decoding stages conducted by a FOR-loop.***

accumulation remains contained throughout the computation (see Fig. 5). As mentioned earlier, the error correction mechanism itself, that is, each interleaved decoding stage, is noisy as well. Thus, the error rate after each decoding stage cannot go to zero. However, the interleaved decoding stages can repeatedly suppress the error and contain error accumulation.

D. Open Problems

Coded computing research has chiefly focused on basic linear operations, namely matrix–vector products or matrix–matrix products. However, a large portion of dense linear algebra routines is solving linear systems, for example, solving linear equations, least-squares problems, or eigenvalue problems. These routines require matrix factorization (e.g., QR factorization or LU factorization) and triangular solve. Although ABFT techniques for these operations have been studied before in the HPC community [72]–[74], there have been few works in coded computing in this direction. One recent work [75] has proposed a new coding strategy for QR decomposition. More broadly, there are exciting opportunities in developing new coding techniques for parallel linear solvers, particularly those that factor in practical concerns in HPC systems.

Another significant open problem that remains is at the intersection of circuits/devices and the proposed strategies. How can models of emerging devices and analog computation engines inform and improve the performance of the proposed strategies? What implementations can perform digital error correction on errors in analog components?

IV. CODED DWARF 2: SPARSE LINEAR ALGEBRA

Sparse linear algebra concerns the problems and methods of manipulating sparse matrices, such as multiplying a sparse matrix to a vector, or performing graph analytics (graphs have sparse matrix representations). Sparse linear algebra has become increasingly important as numerous data sets for machine learning applications are very sparse (e.g., user ratings on products). However, coding sparse matrices using dense codes will make them dense, and increasing computation and communication costs substantially. In this section, we will focus on recent progress that deals with one important sparse linear algebra problem: solving a linear system $\mathbf{Ax} = \mathbf{y}$ in which $\mathbf{A}$ is sparse, which naturally leads (as discussed in [76]) to important problems such as leading eigenvector computations, directly useful in singular value decomposition (SVD) and PCA computation. Note that the system model we consider in this section is large-scale distributed systems and we tackle node failures. Extending the techniques for errors at the gate/device level and developing coding techniques for general sparse linear algebra operations (e.g., sparse matrix–matrix products) on the whole remain open.

A. Sparse Linear Systems: The Importance of the Problem and Techniques in the Literature

Solving a sparse linear system has a variety of applications, including web data analysis [77]–[79], semi-supervised learning [80], partial differential equations [81], [82], circuit simulations [83], power grids [84], and finite element analysis [85]. The connection of sparse linear systems with coding is first mentioned in [86], where a sparse equiangular tight frame (ETF) is used to partially maintain the data sparsity. The proposed technique in [87] encodes the data before solving the linear system, and is thus beyond the scope of coded matrix multiplication. Another line of works of Yang *et al.* [76], Haddadpour *et al.* [88], and Yang *et al.* [89], [90] treats the problem of solving linear systems as repeated matrix–vector multiplications, that is, power method [91], and hence coding can be done across multiple iterations. Since sparse matrices have much larger matrix size than dense matrices for the same data size, ordinary matrix operations such as full-spectrum SVD and matrix inverse are hard to realize. A standard family of techniques to deal with large-scale sparse linear systems originates from different variations of the power method, for example, the widely used Krylov-subspace methods. A comparison of the Krylov-subspace methods and the ordinary power method is given in [92] with an example of solving the PageRank equation [77].

B. Coded Sparse Linear Systems: The "Substitution" Technique

Before we explain the coded computing technique for solving sparse linear systems, we want to discuss a fundamental challenge in coded computation for sparse matrix–vector multiplications. Traditional codes require dense linear combinations of input data. For example, the encoding of a typical coded computing scheme requires multiplying the data matrix with a dense generator matrix $\mathbf{G}$ [see (1)]. However, if we take dense linear combinations of sparse matrices, it can significantly increase the number of nonzero elements. For example, consider

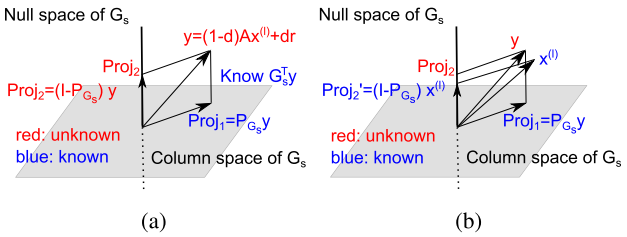

Fig. 6. ***Key ideas in substitute decoding. (a) Finding the maximal information of the vector y by projecting it onto the column space of G_s. (b) Approximating the unknown part of y using the result from the last iteration, $x^{(l)}$.***

computing $\mathbf{Ax}$ with a sparse matrix $\mathbf{A}$ partitioned evenly into ten submatrices, each stored in different distributed nodes. When we introduce redundancy by coding, one linear combination of the ten submatrices using a dense code can lead up to ten times higher storage cost. This problem is unique for sparse data matrices because linearly combining a completely dense matrix does not increase the density. Thus, in practice, for sparse coded matrix multiplications, we are likely to be limited to using extremely sparse codes with small row-weights[5] of their generator matrix $\mathbf{G}$ (ideally a small constant that is independent of the code length). However, fundamental limits on sparse codes for coded computing show that the number of tolerable faults is linearly proportional to the row-weight of the generator matrix [34], [49], [50], [64], [65], [93]. Thus, using a sparse code is not a sufficient solution either. The limitation of using sparse codes can be intuitively described as follows. Suppose the desired result $\mathbf{y}$ is in an encoded form $\mathbf{G}^\top\mathbf{y}$, where $\mathbf{G}$ is the sparse generator matrix. If the row-weight of the generator matrix $\mathbf{G}$ is a small constant, a worst case fault can easily erase all the columns of $\mathbf{G}$ which have a nonzero element on a particular row (with index i). Then, the remaining matrix, denoted by $\mathbf{G}_s$, becomes singular because the submatrix is all-zero on the ith row. This will result in decoding failure as obtaining $\mathbf{y}$ from $\mathbf{G}_s^\top\mathbf{y}$ is not possible.

1) Substitute Decoding for Sparse Linear Systems: We now discuss the recently developed techniques [76], [90] to boost the fault tolerance of extremely sparse codes for linear systems. The problem considered in [76] and [90] is solving sparse linear systems, which is one of the primary applications of sparse matrix multiplications. In particular, we take the famous PageRank equation as an example [77]:

$$\mathbf{x} = (1-d)\mathbf{Ax} + d\mathbf{r}. \tag{2}$$

Here, $\mathbf{A}$ is a sparse column-normalized graph adjacency matrix, and d is the "teleport probability" with which the random walk modeled by PageRank restarts from the initial distribution $\mathbf{r}$. The PageRank equation is solved using the power iterations

$$\mathbf{x}^{(l+1)} = (1-d)\mathbf{Ax}^{(l)} + d\mathbf{r} \tag{3}$$

until $\mathbf{x}^{(l)}$ converges to the fixed point of (2). The computation of $\mathbf{Ax}^{(l)}$ can be done using the standard master–worker setup, in which the sparse matrix $\mathbf{A}$ is partitioned into several row blocks $\mathbf{A} = \left[\mathbf{A}_1^\top \mathbf{A}_2^\top \ldots \mathbf{A}_P^\top\right]^\top$ and each worker machine computes one $\mathbf{A}_i\mathbf{x}^{(l)}$. A straightforward way of coding power iteration is linearly combining the submatrices $\mathbf{A}_i$'s so that the result $\mathbf{y} = (1-d)\mathbf{Ax}^{(l)} + d\mathbf{r}$ is also in a coded form $\mathbf{G}^\top\mathbf{y}$. However, recall that choosing $\mathbf{G}$ to be either dense or sparse comes with downsides (as discussed above).

To overcome this issue, two key ideas were introduced [76]. The first key idea is that even when the partial encoding matrix $\mathbf{G}_s$ (remained after deleting the columns due to failures or stragglers), as mentioned above, is singular, instead of declaring the decoding failure of obtaining $\mathbf{y}$ from $\mathbf{G}_s^\top\mathbf{y}$, we can "maximally" invert $\mathbf{G}_s$ by using its pseudo-inverse, which gives $\mathbf{P}_{\mathbf{G}_s}\mathbf{y}$, in which $\mathbf{P}_{\mathbf{G}_s}$ is the projection onto the column space of $\mathbf{G}_s$. This procedure can maximally preserve the information contained in the successful computations $\mathbf{G}_s^\top\mathbf{y}$. However, this procedure results in information loss of $\mathbf{y}$ because the projection $\mathbf{P}_{\mathbf{G}_s}$ is not full-rank. To get back the remaining part $(\mathbf{I}-\mathbf{P}_{\mathbf{G}_s})\mathbf{y}$, the second key idea is to use the "side information" $\mathbf{x}^{(l)}$ as a proxy of the unknown $\mathbf{y}$. That is to say

$$\mathbf{x}^{(l+1)} = \mathbf{P}_{\mathbf{G}_s}\mathbf{y} + (\mathbf{I}-\mathbf{P}_{\mathbf{G}_s})\mathbf{x}^{(l)}. \tag{4}$$

The intuition is that for the iterative computations given in (2), the intermediate result $\mathbf{y} = (1-d)\mathbf{Ax}^{(l)} + d\mathbf{r}$ and $\mathbf{x}^{(l)}$ get closer to each other as $\mathbf{x}^{(l)}$ converges to the fixed point of (2). Thus, substituting $\mathbf{x}^{(l)}$ in the equation (which is what the authors call "substitute decoding") can maximally recover the required result $\mathbf{y}$ even when the exact computation of $\mathbf{y}$ is unavailable due to failed nodes. These key ideas are illustrated in Fig. 6. In [76], it is shown that substitute decoding for sparsely coded linear systems achieves almost the same convergence rate as the noiseless systems even if there are a large number of random (transient) failures (see Fig. 7). It is also shown that this technique can generalize to the computation of leading eigenvectors of a large and sparse matrix, and the computation of gradient vectors (see [77, Sec. V-B–V-D]).

C. Open Problems

The broad issue of which analog engines could be implemented to support sparse linear algebra remains open, in part because sparse matrices are often stored in compressed forms. Besides this open problem, there is a pending question in designing codes for sparse linear

[5]The row-weight of a generator matrix is defined as the number of nonzeros on each row of the matrix.

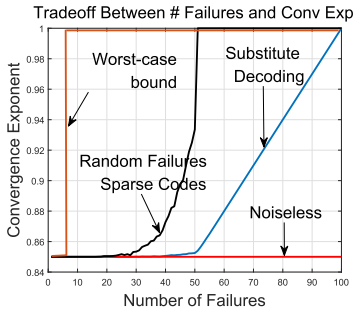

Fig. 7. ***Convergence exponent of substitute decoding (blue line) is close to the optimal value (red line) even when the number of failures is large. Substitute decoding significantly improves the failure tolerance when the failures are random.***

algebra due to the limitation of sparse codes: they provide limited error resilience for worst case failures, especially when the storage overhead is crucial. For example, if one imposes the restriction that the coded data overhead cannot exceed 30%, even the sparsest codes may fail to satisfy the requirement. We propose two different directions we can explore to address this issue.

1) One may consider approximate computation instead of exact computation. For example, the recent works on coded sketching provide an alternative way to address the sparse linear algebra problem [94], [95].
2) The other direction is more ambitious. One can try to design coded schemes that are beyond making linear combinations of the submatrices, and thus bypass the difficulty of using sparse codes. The coded schemes should introduce sufficient redundancy to the data, while essentially maintaining the original sparsity of the data.

V. CODED DWARF 3: SPECTRAL METHODS

A. Spectral Methods

Spectral methods refer to Fourier representations and related operations [such as the fast Fourier transform (FFT)], which convert data into frequency domain. Typically, spectral methods use multiple stages of a butterfly network. When deployed in a distributed way, the butterfly network combines multiply-add operations with a specific pattern of data permutation, using all-to-all communication in some stages and being strictly local in others [28]. FFT operations are widely used in signal processing, and are a valuable tool to speed up scientific computing such as solving differential equations with FFT acceleration [96], [97]. One very important application of FFT is computing convolutions.[6] When convolving two vectors of length N, a brute force approach requires $O(N^2)$ computations. However, one can perform convolution by first converting the vectors into frequency domain, performing element-wise multiplication in the frequency-domain, and then converting back to the original domain. This FFT-based method requires $O(N \log N)$ operations instead of $O(N^2)$.

In this section, we will consider computing an N-point DFT:

$$\mathbf{z} = \mathbf{F}_N \mathbf{x} \tag{5}$$

where $\mathbf{x}$ is an input data vector of length N, $\mathbf{F}_N$ is an N-by-N DFT matrix (ω_N: the Nth root of unity) represented as

$$\mathbf{F}_N = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N-1} & \cdots & \omega_N^{(N-1)^2} \end{bmatrix} \tag{6}$$

and $\mathbf{z}$ is a length-N vector of the Fourier transform of $\mathbf{x}$. We assume that N is large, and the data cannot be stored in one processor. Using the Cooley–Tukey technique [100], the computation of (5) can be broken down into smaller FFTs of size N_1 and N_2 where $N = N_1\ N_2$

$$\begin{aligned} z_k &= \sum_{n=0}^{N-1} \omega_N^{nk} x_n \\ &= \sum_{n_1=0}^{N_1-1} \omega_{N_1}^{n_1 k_1} t_{n_1,k_2} \sum_{n_2=0}^{N_2-1} \omega_{N_2}^{n_2 k_2} x_{n_2 N_1 + n_1} \end{aligned} \tag{7}$$

where $k = k_1\ N_2 + k_2$, $k_1 = 0, \ldots, N_1 - 1$, and $k_2 = 0, \ldots, N_2 - 1$. The terms t_{n_1,k_2}'s are called twiddle factors which are equal to $\omega_N^{k_2\ n_1}$. For simplicity, we will focus on 1-D FFT here, but note that the coding strategies discussed here easily extend to multidimensional FFTs [101], [102].

B. Coded Distributed FFT Strategies

We introduce two recent works on applying the idea of coded computing to distributed FFT algorithms which build up on ABFT techniques for FFT [103]–[106]. One is under a master–worker setup [102] and the other is under a master-less (i.e., decentralized) setup [101]. While these works largely focus on node failure in distributed computing, we will also discuss how these ideas can be extended to analog engines or FFT accelerator circuits.

To utilize (7), let us first rearrange the input data x into a matrix form X

$$\begin{aligned} \mathbf{X} &= \begin{bmatrix} x_1 & x_{N_1+1} & \cdots & x_{(N_2-1)N_1+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N_1} & x_{2\ N_1} & \cdots & x_{N_1\ N_2} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}_1^{(\text{row})} \\ \vdots \\ \mathbf{x}_K^{(\text{row})} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^{(\text{col})} & \cdots & \mathbf{x}_K^{(\text{col})} \end{bmatrix}. \end{aligned}$$

[6] Computing convolutions directly without using FFT was also studied in the coded computing literature. See [54], [98], and [99].

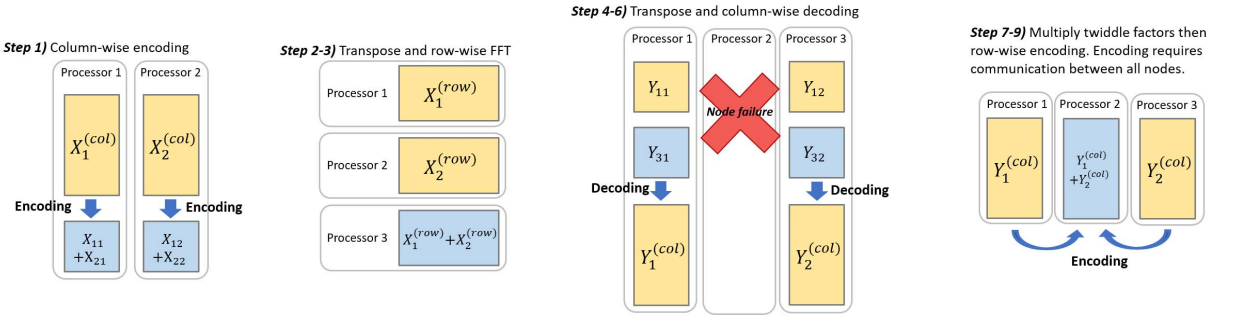

Fig. 8. ***Encoding and decoding steps of masterless coded FFT algorithm with an example of $P = 3, K = 2$ [101, Fig.1].***

Then, computing FFT can be thought of as first computing N_2-point FFT on the row vectors of X, then computing N_1-point FFT on the column vectors. Up to the first N_2-point FFT computation, the two strategies [101], [102] follow the same coding scheme. In both schemes, the input data are encoded as follows:

$$\tilde{\mathbf{x}}_i^{(\text{row})} = g_{i1}\mathbf{x}_1^{(\text{row})} + \cdots + g_{iK}\mathbf{x}_K^{(\text{row})} \tag{8}$$

where $\mathbf{G}^T = [g_{ij}]_{i=1,\ldots,P, j=1,\ldots,K}$ is an encoding matrix of a (P, K)-MDS code. Then, each node gets the encoded data $\tilde{\mathbf{x}}_i^{(\text{row})}$ and performs N_2-point FFT on the encoded data. Due to the linearity, this guarantees that

$$\tilde{\mathbf{y}}_i^{(\text{row})} = g_{i1}\mathbf{y}_1^{(\text{row})} + \cdots + g_{iK}\mathbf{y}_K^{(\text{row})} \tag{9}$$

where $\mathbf{y}_1^{(\text{row})}, \ldots, \mathbf{y}_K^{(\text{row})}$ result from the first N_2-point FFT.

Now, let us discuss how the two strategies differ after this.

1) *Master–Worker Coded FFT:* In the master–worker setup, the master node waits for the first successful K outputs among $\tilde{\mathbf{y}}_1^{(\text{row})}, \ldots, \tilde{\mathbf{y}}_P^{(\text{row})}$. After receiving the computation output from K successful workers, the master decodes these outputs to recover $\mathbf{y}_1^{(\text{row})}, \ldots, \mathbf{y}_K^{(\text{row})}$. Then, the master node carries out the remaining computation: multiplying twiddle factors, and performing the N_1-point FFT.
   Limitations of this strategy are: 1) N can be very large in scientific computing, and the assumption that there is a powerful master node that can store and process the entire vector can be unrealistic and 2) depending on the choice of N_1, computational load at the master node can be as big as the computational load at each worker node, which defeats the purpose of distributed computing.
2) *Master-Less Coded FFT:* In the master-less setup, the second FFT, that is, N_1-point FFT in (7), is performed at distributed nodes as well. After the first N_2-point FFT, the distributed nodes wait until there are K successful workers. Then the successful nodes perform all-to-all communication with each other to transpose their output. These nodes perform decoding locally to recover $\mathbf{y}_1^{(\text{col})}, \ldots, \mathbf{y}_K^{(\text{col})}$. Now, another step of encoding is performed to protect the second FFT step from faults. K nodes that possess $\mathbf{y}_1^{(\text{col})}, \ldots, \mathbf{y}_K^{(\text{col})}$ communicate to the remaining $P-K$ nodes to encode parity symbols. After encoding, all the nodes perform N_1-point FFT on the encoded data. These steps are summarized in Fig. 8.
   While the strategy in [101] overcomes the limitations of master–worker coded FFT, the real challenge is to ensure that communication overhead of distributed encoding and decoding does not exceed the built-in communication cost of the FFT algorithm itself. Jeong *et al.* [101] showed that if we use systematic MDS codes with a very small number of parity nodes, that is, $P - K = o(\log K)$, the communication overhead can be amortized.

C. Open Problems

We believe that the two different coded FFT methods proposed for distributed computing systems can map to connecting multiple FFT accelerators [107]–[109]. As the FFT algorithm is vital for DSP applications and image processing (e.g., convolutions in neural networks [110]) that are being implemented on low-energy embedded systems, there will be continued efforts in building energy-optimized FFT circuits with emerging technologies. The master–worker and master-less coded FFT methods can apply to computing a large-scale FFT (e.g., 1 million point FFT) by connecting small FFT accelerators with unreliability with or without a central controller. Expanding the current results that largely remain theoretical to experimental validation on such systems would be an intriguing future direction.

VI. CODED DWARF 4: MAPREDUCE

A. MapReduce

MapReduce is a widely used framework in the large-scale data processing. It has two phases, "map," and "reduce." In the map phase, input data is split into independent chunks and sent to distributed nodes. At distributed nodes, key/value pairs are processed locally to generate

a set of intermediate key/value pairs. The second phase reduces the returned values from all the nodes into a summarized result by merging intermediate values associated with the same intermediate key. While not exactly in the original seven dwarfs of computation, the "Monte Carlo" dwarf was generalized to MapReduce since the communication and computation pattern of MapReduce generalizes that of Monte Carlo.

In this section, we introduce the coded MapReduce (CMR) strategy [111]. The CMR strategy does not directly add resilience to errors or faults. Instead, the main purpose of applying coding in the CMR strategy is to reduce communication cost. Although this is not perfectly aligned with the core theme of this article, we include the CMR strategy because coding for fault tolerance in MapReduce computations is broadly unaddressed and this article can inspire future work in this direction. Also, on a system where more unreliabilities are present in communication than computation, the CMR strategy can be thought of as a way to mitigate unreliability. Recently, Li *et al.* [33] proposed a unified framework that encapsulates both communication cost reduction and fault tolerance, generalizing results in [32], but this was only limited to matrix–vector multiplication, which was extensively discussed in Section III-A.

B. Coded MapReduce

Between the map and the reduce phase, "data shuffling" is required to rearrange data so that the data with the same intermediate key value can be located in the same worker machine. Often, this data shuffling operation is the bottleneck of MapReduce computations. For example, in Facebook's Hadoop cluster 33% of the total time is consumed by data shuffling, and in self-join applications on Amazon EC2, 70% of the total time is consumed by data shuffling. The core idea of the CMR strategy is to add redundant computations at each worker node to reduce the amount of data that has to be communicated during the shuffle stage. In other words, the CMR strategy leverages the tradeoff between computation and communication.

A Simple Example: Let us explain the CMR strategy through a simple example given in Fig. 9 where we use three nodes to compute three different functions on six input files. Three different shapes (blue triangle, green square, and red circle) represent three different output functions, and we use numbers to denote six different input files. During the data shuffle stage, we want to send all the values associated with the same output function to the same node, that is, red circle outputs to node 1, green squares to node 2, and blue triangles to node 3.

In the uncoded strategy depicted in Fig. 9(a), each input file is located in only one server. For data rearrangement before the reduce phase, each node has to send two of its outputs to the other nodes. Thus, four intermediate values should be communicated from each node, and the total of 12 values need to be communicated. In the coded strategy in Fig. 9(b), each input file is present in two nodes.

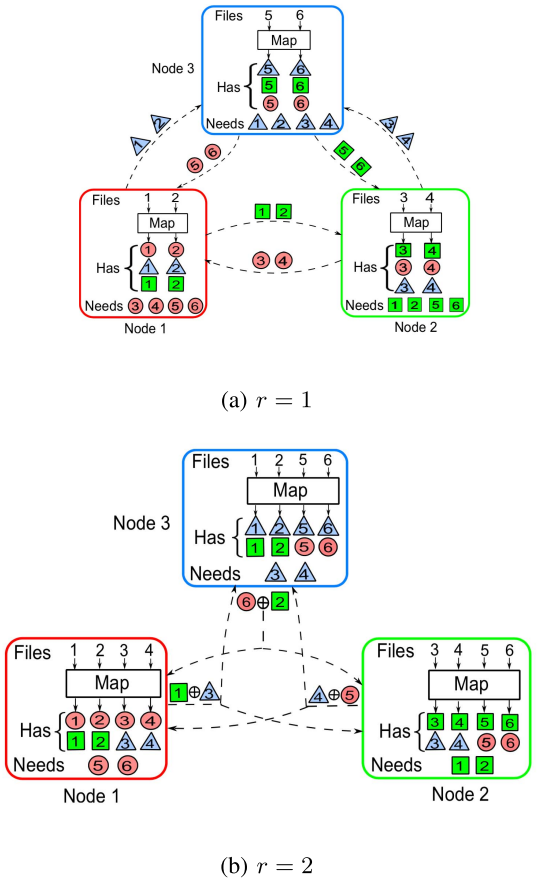

Fig. 9. ***Example of (a) uncoded and (b) CMR. Different shapes represent different output functions, and we use numbers to denote different input files (a) r = 1. (b) r = 2 [112, Fig.1].***

Now, if we do not leverage coding, each node needs two more intermediate values to proceed to the reduce phase. This requires $2 \times 3 = 6$ values to be communicated in total. However, by leveraging coding, we can further reduce this communication load. As illustrated in Fig. 9(b), by sending the XOR of the intermediate values, communication can be reduced to multicasting only three values, instead of six. In fact, Li *et al.* [111] have shown that when we replicate the same computation r times, the CMR strategy reduces the communication load by $1/r$ compared to the uncoded communication load.

C. Experimental Results

Gains achieved by the CMR strategy are experimentally quantified [112] using the TeraSort benchmark [113], which is a popular benchmark to evaluate the performance of MapReduce. The authors evaluated coded Terasort implementation on Amazon EC2, and showed a significant speed-up of coded TeraSort over uncoded TeraSort. Experimental results of sorting 12 GB of data with 16 workers are summarized in Table 1. Note that theoretically, communication is reduced by r times, but the speedup in the experiments is less than r due to increased computation cost and the absence of network-layer multicast support in EC2.

Table 1 Summary of Experimental Results [112] for Sorting 12 GB of Data With $K = 16$ Workers

	Map (s)	Shuffle (s)	Reduce (s)	Total (s)	Speedup
Uncoded TeraSort	1.86	945.7	10.47	961.3	
Coded TeraSort ($r = 3$)	6.03	412.2	13.05	445.6	2.16x
Coded TeraSort ($r = 5$)	10.84	222.8	14.40	283.3	3.39x

D. Open Problems

There exists a chasm between the CMR strategy and the unified coding framework for MapReduce [33]. The CMR strategy works for any function but it does not provide fault tolerance. The coding strategy in [33] provides fault tolerance but works only for matrix–vector products, which is extensively studied beyond the MapReduce context. Coded computing techniques are needed that can be applied universally (similar to the CMR strategy), or to a broader set of problems, to provide resilience against faults/errors during computation.

VII. APPLICATION OF CODED DWARFS: TRAINING A DNN

DNNs are rapidly becoming an important tool in modern computing applications. Training a DNN can be very compute-intensive and memory-demanding as we use a larger model. To distribute the task of training a DNN, people often adopt either data-parallel or model-parallel architecture. In data parallelism, different nodes store and train a replica of the entire DNN on different pieces of data, and a central parameter server combines gradient updates from all the nodes to train a central replica of the DNN. In model parallelism, different parts of a single DNN are distributed across multiple nodes.

1) Coding Data-Parallel Training Using "Gradient Coding": Coding for data-parallel training is examined in [48] where the problem is viewed as a matrix multiplication problem. Here, the matrix $\mathbf{A}$ is fixed and known in advance and is required to be multiplied with another vector or matrix (the gradients) generated in real-time. In fact, gradient coding [48]–[51] uses short-dot codes (described in Section III-A2) at its core.[7] The data set is divided into N partitions. At each iteration of training, the master node requires the sum of the gradients evaluated on all the data partitions. Thus, the matrix $\mathbf{A}$ here is essentially a row vector $[1, 1, \ldots, 1]_{1\times N}$, and it is to be multiplied with a matrix whose rows are gradients from a different partition of the data set, that is

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1^{(\text{row})} \\ \mathbf{x}_2^{(\text{row})} \\ \cdots \\ \mathbf{x}_N^{(\text{row})} \end{bmatrix}.$$

[7]The two results, gradient coding [48] and short-dot codes [34], were arrived at simultaneously.

The goal is to generate a set of $P(= N)$ sparse vectors, one for each node, such that any subset of size K can linearly span $[1, 1, \ldots, 1]_{1\times N}$. The ith node computes a dot-product of its corresponding sparse vector and $\mathbf{x}$, and sends the result to the master node. Because these generated N vectors are sparse, the ith node only stores and computes gradients on the data partitions indexed in the support set of its corresponding sparse vector, and not on the entire data set which reduces the computation at each node. The master node waits for any K nodes to finish as they are sufficient to compute the required gradient and thus tolerates stragglers.

2) Model-Parallel Training: The problem of coded DNN training requires making all the steps of DNN training resilient to soft errors (bit flips during computation that produce erroneous outputs) under the constraint that each node has limited storage, that is, it can store only a fixed fraction of each of the weight matrices being trained. To compute the gradients, at each layer, we multiply the weight matrix $\mathbf{A}$ with an operand from the right side during the feedforward stage and again with an operand from in the backpropagation stage, along with other low-complexity operations such as nonlinear operations, Hadamard product, etc. Finally, in the update stage, the weight matrices at each layer are all updated.

Dutta *et al.* [39], [40], and [47] studied training a DNN under the masterless setting and identified three challenges that have to be addressed in applying coded computing to this problem: 1) prohibitively large overhead of coding the weight matrices in each layer of the DNN at each iteration; 2) nonlinear operations during the feedforward stage when propagating from one layer to the next, which are incompatible with linear coding; and 3) absence of an error-free master node, requiring us to architect a fully decentralized implementation.

Toward addressing these additional challenges, carefully weaving MDS codes (see [39]) or generalized PolyDot codes (see [40] and [47]) into the operations of DNN training was proposed so that an initial encoding of the weight matrices is maintained across the updates at each iteration. To do so, at each iteration each node locally encodes much smaller matrices consisting of NB elements instead of the large matrix $\mathbf{A}$ of N^2 elements where B is the mini-batch size, adding negligible overhead. In particular, for the case of $B = 1$, this simply reduces to encoding vectors instead of matrices, which is much cheaper in terms of computational complexity. Next, we circumvent the nonlinear activation between layers by coding each layer separately. Lastly, we also enable a fully decentralized implementation by making each node as a functional replica of the master node. Each node performs low-complexity operations that were usually done in a master node such as aggregating the partial computation results, detecting/correcting errors, and subsequent computing and encoding to generate inputs for the next stage of matrix-multiplication. Simulation results in [39] show that a unified MDS coded

DNN training strategy attains a better accuracy-runtime tradeoff as compared to an uncoded strategy that simply ignores soft errors or a replication based strategy.

This article is a significant step in the design of biologically inspired neural networks with error resilience that could hold the key to significant improvements in efficiency and reduction of energy consumption during neural network training. Thus, these results could be of broader scientific interest to communities like HPC, neuroscience as well as neuromorphic computing.

VIII. DISCUSSION AND FUTURE DIRECTIONS

In this article, we reviewed coded computing techniques to build fault tolerance into four of the seven dwarfs. Examining coded computing techniques for the remaining dwarfs (including dwarfs added later, see Asanovic *et al.* [28]) is an important future direction.

Integrating these techniques closely with design of emerging devices and systems is perhaps the most imperative future direction. For instance, in an unpublished work with Ning Wang and Pop (the authors of [23]), we developed the concept of "analog computation flags." These flags are small computations outputs that indicate the confidence an analog engine has in its own output, based on its modeling of input dependent errors (as discussed in [23], where this modeling is done for graphene-based dot-product analog engines), and are attached to the original analog computation engine which computes the target function. Such novel systems, when implemented in association with analog compute engines, could help alleviate the system-level fault-tolerance problem because it can help identify which nodes have erroneous outputs, and discard those outputs from decoding (in coding-theory parlance, they reduce errors to erasures).

A caveat in coded computing is that the codes developed for computing have to apply to real-valued computations, unlike classical ECCs that apply to finite field computations. An important consequence is that often, scalability is also limited by numerical precision, especially when polynomial-based methods are used and decoding requires polynomial interpolation (see [62] and [93]). By developing novel code constructions using orthogonal polynomials in numerical analysis and random matrix-based techniques, recent works [114]–[117] have significantly improved upon the codes of [32], [34], [35], [48], [55], and [118] in terms of numerical stability for matrix–vector, matrix–matrix products and polynomial evaluations. As new code constructions appear in the form of solutions to the open problems posed in this article, and as new hardware technologies emerge with varying precision capabilities, the study of stability issues alongside code constructions will form an interesting and crucial research thread toward enabling scalability in practice.

In storage systems, we have seen many successful cases where research collaborations between system/device designers and information/coding theorists generated not only practical values but also theoretical advances. For example, new classes of codes were developed for flash memory [119], [120] and resistive memory [121] which are designed to combat device-specific fault patterns and vulnerabilities. Codes designed to overcome the constraints of today's distributed storage systems [122]–[124] are now widely adopted in practice. We believe that the same can be achieved for computing systems and devices. By thinking beyond traditional fault tolerance techniques and designing codes based on the understanding of the system limitations and device-specific characteristics, newly emerging computing systems can be made robust and resilient with minimal overhead. ■

REFERENCES

[1] M. Duranton, K. De Bosschere, C. Gamrat, J. Maebe, H. Munk, and O. Zendra, *The HiPEAC Vision 2017*. HiPEAC Network of Excellence, 2017.

[2] X. Jia *et al.*, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," 2018, *arXiv:1807.11205*. [Online]. Available: http://arxiv.org/abs/1807.11205

[3] M. Rännar. (2019). *Parallel Computer Systems, Lecture 2*. [Online]. Available: https://www8.cs.umu.se/kurser/5DV011/VT10/slides/pds02.pdf

[4] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2012, pp. 15–28.

[5] W. S. Evans and L. J. Schulman, "Signal propagation and noisy circuits," *IEEE Trans. Inf. Theory*, vol. 45, no. 7, pp. 2367–2373, 1999.

[6] Y. Polyanskiy and Y. Wu, "Dissipation of information in channels with input constraints," 2014, *arXiv:1405.3629*. [Online]. Available: http://arxiv.org/abs/1405.3629

[7] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proc. Int. Conf. High Perform. Comput. Comput. Sci.* Berlin, Germany: Springer, 2010, pp. 1–25.

[8] P. Kogge *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," Defense Adv. Res. Projects Agency Inf. Process. Techn. Office, Arlington County, VA, USA, Tech. Rep., 2008.

[9] K. Ferreira *et al.*, "Evaluating the viability of process replication reliability for exascale systems," in *Proc. Int. Conf. for High Perform. Comput., Netw., Storage Anal. (SC)*, 2011, pp. 1–12.

[10] A. Benoit, T. Herault, V. L. Fèvre, and Y. Robert, "Replication is more efficient than you think," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA, 2019, pp. 1–14.

[11] Fujitsu Limited Press Release. (2019). *Fujitsu Begins Shipping Supercomputer Fugaku*. [Online]. Available: https://www.fujitsu.com/global/about/resources/news/press-releases/2019/1202-01.html

[12] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2017, pp. 1–12.

[13] A. Geist, "How to kill a supercomputer: Dirty power, cosmic rays, and bad solder," *IEEE Spectr.*, vol. 10, pp. 2–3, Nov. 2016.

[14] A. Harlap *et al.*, "Addressing the straggler problem for iterative convergent parallel ML," in *Proc. 7th ACM Symp. Cloud Comput. (SoCC)*, 2016, pp. 98–111.

[15] E. Krevat, J. Tucek, and G. R. Ganger, "Disks are like snowflakes: No two are alike," in *Proc. HotOS*, 2011, pp. 1–4.

[16] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. 3rd ACM Symp. Cloud Comput. (SoCC)*, 2012, pp. 1–13.

[17] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds," in *Proc. ACM Symp. Cloud Comput.*, 2012, pp. 1–7.

[18] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using Mantri," in *Proc. USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2010, vol. 10, no. 1, p. 24.

[19] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 185–198.

[20] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2003, p. 55.

[21] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2006, pp. 1–12.

[22] R. M. Roth, "Fault-tolerant dot-product engines,"

IEEE Trans. Inf. Theory, vol. 65, no. 4, pp. 2046–2057, Apr. 2019.

[23] N. C. Wang, S. K. Gonugondla, I. Nahlus, N. R. Shanbhag, and E. Pop, "GDOT: A graphene-based nanofunction for dot-product computation," in *Proc. IEEE Symp. VLSI Technol.*, Jun. 2016, pp. 1–2.

[24] I. Nahlus, E. P. Kim, N. R. Shanbhag, and D. Blaauw, "Energy-efficient dot product computation using a switched analog circuit architecture," in *Proc. Int. Symp. Low power Electron. Design (ISLPED)*, 2014, pp. 315–318.

[25] B. E. Boser, E. Sackinger, J. Bromley, Y. Le Cun, and L. D. Jackel, "An analog neural network processor with programmable topology," *IEEE J. Solid-State Circuits*, vol. 26, no. 12, pp. 2017–2025, Dec. 1991.

[26] M. Hu *et al.*, "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in *Proc. ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[27] P. Colella, "Defining software requirements for scientific computing," Defense Adv. Res. Projects Agency High Productiv. Comput. Syst., Arlington County, VA, USA, Tech. Rep., 2004.

[28] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from Berkeley," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2006-183, 2006.

[29] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-100, no. 6, pp. 518–528, Jun. 1984.

[30] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, no. 10, pp. 1304–1308, Oct. 1990.

[31] Y.-H. Choi and M. Malek, "A fault-tolerant FFT processor," *IEEE Trans. Comput.*, vol. C-37, no. 5, pp. 617–621, May 1988.

[32] 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, Mar. 2018.

[33] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, Dec. 2016, pp. 1–6.

[34] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2092–2100.

[35] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," in *Proc. IEEE 55th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, 2017, pp. 1264–1270.

[36] U. Sheth *et al.*, "An application of storage-optimal MatDot codes for coded matrix multiplication: Fast k-nearest neighbors estimation," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 1113–1120.

[37] Q. Yu, M. Ali Maddah-Ali, and A. Salman Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," 2018, *arXiv:1801.07487*. [Online]. Available: http://arxiv.org/abs/1801.07487

[38] Y. Yang, P. Grover, and S. Kar, "Computing linear transformations with unreliable components," *IEEE Trans. Inf. Theory*, vol. 63, no. 6, pp. 3729–3756, Jun. 2017.

[39] S. Dutta, Z. Bai, T. Meng Low, and P. Grover, "CodeNet: Training large scale neural networks in presence of soft-errors," 2019, *arXiv:1903.01042*. [Online]. Available: http://arxiv.org/abs/1903.01042

[40] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, "A unified coded deep neural network training strategy based on generalized PolyDot codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2018, pp. 1585–1589.

[41] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*. Princeton, NJ, USA: Princeton Univ. Press, 1956, pp. 329–378.

[42] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biol.*, vol. 52, nos. 1–2, pp. 99–115, Jan. 1990.

[43] S. C. Phillips, V. Engen, and J. Papay, "Snow white clouds and the seven dwarfs," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, Nov. 2011, pp. 738–745.

[44] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," in *Proc. Symp. Appl. Specific Processors*, Jun. 2008, pp. 101–107.

[45] W. Dally, "High-performance hardware for machine learning," presented at the NIPS Tutorial, Montreal, QC, Canada, Dec. 2015.

[46] W. Ryan and S. Lin, *Channel Codes: Classical and Modern*. Cambridge, U.K.: Cambridge Univ. Press, 2009.

[47] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, "A unified coded deep neural network training strategy based on generalized PolyDot codes for matrix multiplication," 2018, *arXiv:1811.10751*. [Online]. Available: http://arxiv.org/abs/1811.10751

[48] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding," 2016, *arXiv:1612.03301*. [Online]. Available: http://arxiv.org/abs/1612.03301

[49] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2017, pp. 3368–3376.

[50] N. Raviv, R. Tandon, A. Dimakis, and I. Tamo, "Gradient coding from cyclic MDS codes and expander graphs," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2018, pp. 4302–4310.

[51] W. Halbawi, N. Azizan-Ruhi, F. Salehi, and B. Hassibi, "Improving distributed gradient descent using Reed–Solomon codes," 2017, *arXiv:1706.05436*. [Online]. Available: http://arxiv.org/abs/1706.05436

[52] G. Suh, K. Lee, and C. Suh, "Matrix sparsification for coded matrix multiplication," in *Proc. 55th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Oct. 2017, pp. 1271–1278.

[53] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2017, pp. 2418–2422.

[54] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: An optimal design for high-dimensional coded matrix multiplication," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 4403–4413.

[55] 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, Jan. 2020.

[56] M. F. Aktas, P. Peng, and E. Soljanin, "Effective straggler mitigation: Which clones should attack and when?" *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 2, pp. 12–14, Oct. 2017.

[57] G. Nannicini, "Straggler mitigation by delayed relaunch of tasks," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 3, pp. 248–248, Mar. 2018.

[58] M. Aliasgari, J. Kliewer, and O. Simeone, "Coded computation against processing delays for virtualized cloud-based channel decoding," *IEEE Trans. Commun.*, vol. 67, no. 1, pp. 28–38, Jan. 2019.

[59] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, "Coded computation over heterogeneous clusters," *IEEE Trans. Inf. Theory*, vol. 65, no. 7, pp. 4227–4242, Jul. 2019.

[60] N. S. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in *Proc. 54th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2016, pp. 954–960.

[61] N. Ferdinand and S. C. Draper, "Hierarchical coded computation," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2018, pp. 1620–1624.

[62] U. Sheth *et al.*, "An application of storage-optimal MatDot codes for coded matrix multiplication: Fast k-nearest neighbors estimation," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 1113–1120.

[63] A. Mallick, M. Chaudhari, and G. Joshi, "Fast and efficient distributed matrix-vector multiplication using rateless fountain codes," in *Proc. ICASSP - IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 8192–8196.

[64] S. Wang, J. Liu, and N. Shroff, "Coded sparse matrix multiplication," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2018, pp. 5139–5147.

[65] S. Wang, J. Liu, N. Shroff, and P. Yang, "Fundamental limits of coded linear transform," 2018, *arXiv:1804.09791*. [Online]. Available: http://arxiv.org/abs/1804.09791

[66] A. Severinson, A. Graell i Amat, and E. Rosnes, "Block-diagonal and LT codes for distributed computing with straggling servers," *IEEE Trans. Commun.*, vol. 67, no. 3, pp. 1739–1753, Mar. 2019.

[67] H. Jeong *et al.*, "3D Coded SUMMA: Communication-efficient and robust parallel matrix multiplication," in *Proc. 26th Int. Eur. Conf. Parallel Distrib. Comput.*, Aug. 2020.

[68] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 D matrix multiplication and lu factorization algorithms," in *Proc. Eur. Conf. Parallel Process.* Berlin, Germany: Springer, 2011, pp. 90–109.

[69] M. Sipser and D. A. Spielman, "Expander codes," *IEEE Trans. Inf. Theory*, vol. 42, no. 6, pp. 1710–1722, Nov. 1996.

[70] S. M. S. T. Yazdi, H. Cho, and L. Dolecek, "Gallager B decoder on noisy hardware," *IEEE Trans. Commun.*, vol. 61, no. 5, pp. 1660–1673, May 2013.

[71] C.-H. Huang, Y. Li, and L. Dolecek, "Gallager b LDPC decoder with transient and permanent errors," *IEEE Trans. Commun.*, vol. 62, no. 1, pp. 15–28, Jan. 2014.

[72] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: A fault tolerant implementation without checkpointing," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 162–171.

[73] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 225–234, Sep. 2012.

[74] A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy," *ACM Trans. Parallel Comput.*, vol. 1, no. 2, pp. 1–28, Feb. 2015.

[75] Q. M. Nguyen, H. Jeong, and P. Grover, "Coded QR decomposition," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, 2020.

[76] Y. Yang, M. Chaudhari, P. Grover, and S. Kar, "Coded iterative computing using substitute decoding," 2018, *arXiv:1805.06046*. [Online]. Available: http://arxiv.org/abs/1805.06046

[77] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Stanford, CA, USA, Tech. Rep. 1999-66, 1999.

[78] T. H. Haveliwala, "Topic-sensitive PageRank," in *Proc. 11th Int. Conf. World Wide Web (WWW)*, 2002, pp. 517–526.

[79] C. Li *et al.*, "Fast computation of SimRank for static and dynamic information networks," in *Proc. 13th Int. Conf. Extending Database Technol. (EDBT)*, 2010, pp. 465–476.

[80] J. Tang, R. Hong, S. Yan, T.-S. Chua, G.-J. Qi, and R. Jain, "Image annotation by k NN-sparse graph-based label propagation over noisily tagged web images," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 2, pp. 1–15, Feb. 2011.

[81] P. Concus, G. H. Golub, and D. P. O'Leary, "A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations," in *Sparse Matrix Computations*. Amsterdam, The Netherlands: Elsevier, 1976, pp. 309–332.

[82] N. Munksgaard, "Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients," *ACM Trans. Math. Softw.*, vol. 6, no. 2, pp. 206–219, Jun. 1980.

[83] R. W. Freund, "Krylov-subspace methods for reduced-order modeling in circuit simulation," *J. Comput. Appl. Math.*, vol. 123, nos. 1–2, pp. 395–421, Nov. 2000.

[84] T.-H. Chen and C. C.-P. Chen, "Efficient large-scale power grid analysis based on preconditioned Krylov-subspace iterative methods," in *Proc. 38th Conf. Design Autom. (DAC)*, 2001, pp. 559–562.

[85] A. S. Gullerud and R. H. Dodds, "MPI-based implementation of a PCG solver using an EBE architecture and preconditioner for implicit, 3-D finite element analysis," *Comput. Struct.*, vol. 79, no. 5, pp. 553–575, Feb. 2001.

[86] C. Karakus, Y. Sun, and S. Diggavi, "Encoded distributed optimization," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2017, pp. 2890–2894.

[87] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 5440–5448.

[88] F. Haddadpour, Y. Yang, M. Chaudhari, V. R. Cadambe, and P. Grover, "Straggler-resilient and communication-efficient distributed iterative linear solver," 2018, *arXiv:1806.06140*. [Online]. Available: http://arxiv.org/abs/1806.06140

[89] Y. Yang, P. Grover, and S. Kar, "Coded distributed computing for inverse problems," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 709–719.

[90] Y. Yang, P. Grover, and S. Kar, "Coding for a single sparse inverse problem," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2018, pp. 1575–1579.

[91] Y. Saad, *Iterative Methods for Sparse Linear Systems*, vol. 82. Philadelphia, PA, USA: SIAM, 2003.

[92] D. Gleich, L. Zhukov, and P. Berkhin, "Fast parallel PageRank: A linear system approach," Yahoo Res., Sunnyvale, CA, USA, Tech. Rep. YRL-2004-038, 2004.

[93] M. Ye and E. Abbe, "Communication-computation efficient gradient coding," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2018, pp. 5606–5615.

[94] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, "OverSketch: Approximate matrix multiplication for the cloud," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 298–304.

[95] T. Jahani-Nezhad and M. Ali Maddah-Ali, "CodedSketch: A coding scheme for distributed computation of approximated matrix multiplication," 2018, *arXiv:1812.10460*. [Online]. Available: http://arxiv.org/abs/1812.10460

[96] R. D. Ryne, "On FFT-based convolutions and correlations, with application to solving Poisson's equation in an open rectangular pipe," 2011, *arXiv:1111.4971*. [Online]. Available: http://arxiv.org/abs/1111.4971

[97] R. W. Hockney, "A fast direct solution of Poisson's equation using Fourier analysis," *J. ACM*, vol. 12, no. 1, pp. 95–113, Jan. 1965.

[98] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2017, pp. 2403–2407.

[99] Y. Yang, P. Grover, and S. Kar, "Fault-tolerant parallel linear filtering using compressive sensing," in *Proc. 9th Int. Symp. Turbo Codes Iterative Inf. Process. (ISTC)*, Sep. 2016, pp. 201–205.

[100] W. M. Gentleman and G. Sande, "Fast Fourier transforms: For fun and profit," in *Proc. Comput. Conf.*, Nov. 1966, pp. 563–578.

[101] H. Jeong, T. M. Low, and P. Grover, "Masterless coded computing: A fully-distributed coded FFT algorithm," in *Proc. 56th Annu. Allerton Conf. Commun., Control Comput. (Allerton)*, Oct. 2018, pp. 887–894.

[102] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded Fourier transform," 2017, *arXiv:1710.06471*. [Online]. Available: http://arxiv.org/abs/1710.06471

[103] 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.

[104] D. L. Tao and C. R. P. Hartmann, "A novel concurrent error detection scheme for FFT networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 2, pp. 198–221, Feb. 1993.

[105] S.-J. Wang and N. K. Jha, "Algorithm-based fault tolerance for FFT networks," *IEEE Trans. Comput.*, vol. 43, no. 7, pp. 849–854, Jul. 1994.

[106] C. Gun Oh, H. Yong Youn, and V. K. Raj, "An efficient algorithm-based concurrent error detection for FFT networks," *IEEE Trans. Comput.*, vol. 44, no. 9, pp. 1157–1162, Sep. 1995.

[107] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Design Autom. Electron. Syst.*, vol. 17, no. 2, pp. 1–33, Apr. 2012.

[108] Q. Guo *et al.*, "3D-stacked memory-side acceleration: Accelerator and system design," in *Proc. Workshop Near-Data Process. (WoNDP) MICRO*, 2014.

[109] S. Hsu *et al.*, "A 2 GHz 13.6 mW 12x9b multiplier for energy efficient FFT accelerators," in *Proc. 31st Eur. Solid-State Circuits Conf. (ESSCIRC)*, 2005, pp. 199–202.

[110] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2016, pp. 26–35.

[111] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," in *Proc. 53rd Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2015, pp. 964–971.

[112] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded TeraSort," in *Proc. Int. Workshop Parallel Distrib. Comput. Large Scale Mach. Learn. Big Data Analytics*, 2017, pp. 389–398.

[113] *Hadoop TeraSort*. Accessed: May 3, 2020. [Online]. Available: https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html

[114] M. Fahim and V. R. Cadambe, "Numerically stable polynomially coded computing," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2019, pp. 3017–3021.

[115] A. Ramamoorthy and L. Tang, "Numerically stable coded matrix computations via circulant and rotation matrix embeddings," 2019, *arXiv:1910.06515*. [Online]. Available: http://arxiv.org/abs/1910.06515

[116] A. M. Subramaniam, A. Heidarzadeh, and K. R. Narayanan, "Random Khatri-Rao-product codes for numerically-stable distributed matrix multiplication," 2019, *arXiv:1907.05965*. [Online]. Available: http://arxiv.org/abs/1907.05965

[117] A. Ramamoorthy, L. Tang, and P. O. Vontobel, "Universally decodable matrices for distributed matrix-vector multiplication," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2019, pp. 1777–1781.

[118] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *Proc. Int. Conf. Artif. Intell. Statist. (AISTATS)*, 2019, pp. 1215–1225.

[119] L. M. Grupp *et al.*, "Characterizing flash memory: Anomalies, observations, and applications," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (Micro-42)*, Dec. 2009, pp. 24–33.

[120] L. Dolecek and F. Sala, "Channel coding methods for non-volatile memories," *Found. Trends Commun. Inf. Theory*, vol. 13, no. 1, pp. 1–128, 2016.

[121] Y. Kim *et al.*, "Locally rewritable codes for resistive memories," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 9, pp. 2470–2485, Sep. 2016.

[122] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A 'Hitchhiker's' guide to fast and efficient data reconstruction in erasure-coded data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 331–342, 2015.

[123] K. Rashmi, N. B. Shah, and K. Ramchandran, "A piggybacking design framework for read-and download-efficient distributed storage codes," in *Proc. IEEE Int. Symp. Inf. Theory Proc. (ISIT)*, Jul. 2013, pp. 331–335.

[124] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Aug. 2010.