Codes for Distributed Computing: A Tutorial

Viveck Cadambe and Pulkit Grover

This article is based on the material presented in a tutorial in the 2017 IEEE International Symposium on Information Theory (ISIT), Aachen Germany in June 2017. We write this article in two parts that mirror the structure of the tutorial presented in ISIT. First we focus on *shared memory emulation*—a classical problem in the area of distributed computing [60, 9]. We give a tutorial-like overview of shared memory emulation and its applications, and then describe relevant coding-theoretic formulations. The high level goal of shared memory emulation is to build algorithms that expose a distributed data storage system with certain desirable properties to external clients. Our description may therefore be viewed as a transition from the area of codes for distributed storage, where algorithmic aspects can often be ignored in coding-related studies, to distributed computing, where algorithmic aspects play a central role in problem formulation and solutions.

Second, we describe how codes can be used to perform reliable computations using unreliable processing elements. Just as sophisticated error-correction techniques that achieve the Shannon capacity have revolutionized communication on unreliable channels, use of error-correction techniques in computing has the potential to revolutionize next generation computing systems. In fact, techniques of replication and error-correction have been used in computing for decades now. In the last few years, information-theory community has contributed significantly to both fundamental limits and achievable strategies in this area. The purpose of this section is to put this recent information-theoretic work in the broader historical perspective of the area, and utilize that to suggest intellectually interesting and important research directions.

We admit that our goal is to not be comprehensive. Instead, this article is intended for graduate students and researchers interested in the area to learn about some of the key ideas, and is by its nature biased by our own interests and perspectives.

1. Part I: Shared Memory Emulation

1.1. Introduction

A read/write memory admits two operations: *write(variablenam.e, value)* and *read(variablename)*. The goal of the *read/write shared memory emulation* problem is to implement a shared read/write memory over a distributed system of processing nodes (Fig. 1). For simplicity[1], we focus here on a single variable and omit the *variablename* parameter. When the same read/write memory is concurrently accessed by multiple *client* nodes—possibly to jointly run some application or perform some task—it is usually referred to as a shared memory.

[1]A reader familiar with distributed systems theory will note that there can be a significant loss of generality in studying just a single variable, in particular, in systems that are not atomic. Specifically, consistency criteria weaker than atomicity are not "composable", so the study of a single variable may not suffice from an algorithmic viewpoint, nonetheless, our description here is restricted to atomic variables and therefore focusing on single variable suffices.

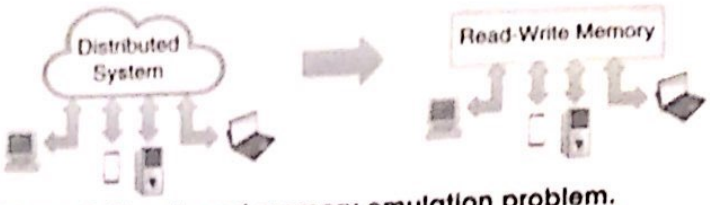

Figure 1. The shared memory emulation problem.

Algorithms that solve the shared memory emulation problem provide theoretical underpinnings of cloud- based data-storage/database services that offer a property called *consistency*, which is explained in Sec. 1.2.1. There are numerous commercial and open source cloud based consistent data storage services such as Amazon Dynamo DB [24], Couch DB [7], Apache Cassandra DB [45], Berkeley DB [65], Redis DB [19].). Consistent data storage services are used in a wide variety of applications such as reservation systems, financial transactions, multi-player gaming [14], online social networks, and fog computing [79, 61]. Modern consistent data storage system design combines distributed computing theoretic approaches that design provably correct algorithms with systems engineering innovation to provide fast read and write operations. The study of shared memory emulation and design of consistent data storage services is an active area of research in distributed computing systems.

In consistent data storage services, the speed of access to data is critical, and therefore such data is commonly cached in relatively expensive high speed memory (RAM or solid state devices), and flushed to the magnetic disks, which is a less expensive, slower medium, only in the case of memory overflow. While data replication is often used in practice to provide fault tolerance, motivated by the need to use memory in the most efficient way possible, erasure coding based algorithms to provide consistent distributed storage services have been studied in the distributed computing theory and systems research communities [42, 27, 15, 16, 25, 53, 52, 71, 96, 20]. The use of erasure coding for such applications poses interesting challenges and research opportunities in information and coding theory, distributed algorithms, and optimization. We first describe some of the key concepts of shared memory emulation. We then describe an information theoretic framework to study the problem, and conclude with open directions of research.

1.2. The Shared Memory Emulation Problem

In this section, we first describe a distributed systems model, and then explain the concept of consistency. We then provide high level descriptions of the shared memory emulation problem and its solutions, where we explain some of the challenges of erasure coding based shared memory emulation.

1.2.1. Distributed Systems Model

The shared memory emulation problem consists of client nodes and server nodes, where the clients nodes may be partitioned in to read and write clients (See Fig. 2). Write clients issue write

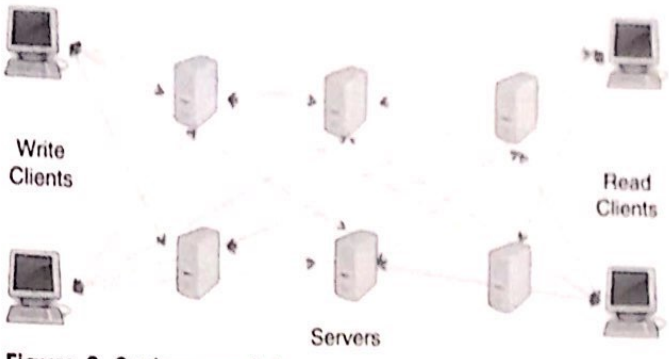

Figure 2. System model.

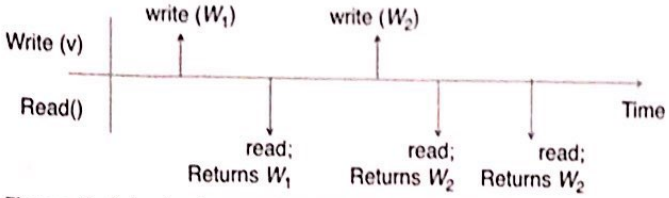

Figure 3. A typical execution of an ideal read-write memory.

operations, and read clients issue read operations[2]. We assume that there is a system with N server nodes and an arbitrary number of client nodes. The maximum number of server nodes that fail is f. The following assumptions distinguish the shared memory emulation model from commonly studied models of distributed storage systems in information theory research.

A1—Arbitrary Asynchrony: The nodes are all connected by (logical) point-to-point links[3] and the topology is assumed to be known to all nodes. The point-to-point links are themselves thought of as asynchronous links which are reliable, but their delay can be arbitrary and unbounded.

A2—Nodes are computing devices: The nodes are not merely storage devices but they are computing devices, and can therefore be used to execute fairly complex, interactive, protocols.

A3—Decentralized Nature: A node is unaware of the current state of any other node, and a node's knowledge of the state of another node is limited to what it can learn from the messages it receives on the incoming links.

In a system with N servers, we aim to design algorithms that operate correctly so long as the number of server failures no larger than some known threshold f. Note that because the system is asynchronous (assumption **A1**), a failed node cannot be distinguished from a very slow node; a node that does not obtain responses from another node cannot figure out if that node has failed or the messages from the node may simply be delayed. Furthermore, when a server node receives the updated data from a write client, it is not automatically aware of whether another node in the system has received this update yet. These aspects can make correct distributed algorithm design in the above model quite interesting and challenging. In practice, server nodes are typically in a cloud storage system, such as a data center, and client nodes are external devices or proxy/leader nodes in the storage system carrying out operations on behalf of an external clients.

[2]The split between read and write clients is a logical split—they can be physically be located on the same node.

[3]This system is often referred to as the *message passing* architecture.

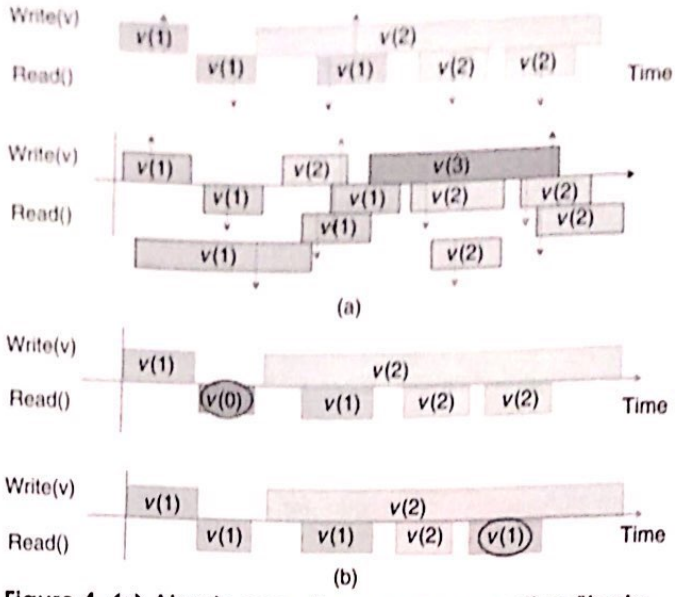

Figure 4. (a) Atomic executions: every operation "looks like" it took place instantaneously at some point in its interval, (b) Executions that are not atomic: we cannot place points in the intervals of operations and make the execution look like that of a correct instantaneous read-write memory.

Concept of atomic consistency: A typical execution of an *ideal* shared read-write memory is presented in Fig. 3. However, the execution of Fig. 3 is too idealistic to model in distributed systems, specifically, read and write operations are seldom instantaneous, but they take time. More specifically, the time scale of operation completion is often comparable to the inter-arrival time of operations; therefore operations may overlap[4].

In systems where operations can overlap (e.g. Fig. 4), it is necessary to carefully define a set of rules that govern the possible outcomes of the write and read operations: such rules are known as *consistency* requirements. Consistency requirements are usually defined to ensure that (a) they are sufficiently relaxed so that they can be implemented in a realistic asynchronous distributed system, and (b) they are strict enough such that the overall execution is, in some sense, indistinguishable from one that could have taken place over an instantaneous read-write memory. For instance, an important requirement is that a read operation obtains the latest (in some sense) write operation. There are several formal ways of defining consistency, each of which is useful depending on the system and applications using the system. Here we give a high level explanation of an important consistency requirement known as *atomic consistency* or simply *atomicity* [54][5].

An execution is said to be atomic if every operation of the execution "looks like" it took place at some point in the interval of the operation. Examples of atomic executions are provided in Fig. 4a. Note that in each execution of Fig. 4a, we can place points in the interval of an operation such that, if we "shrink" the operation to that point, the overall execution will look like a correct execu-

[4]Operation overlap can be avoided through protocols that implement a *lock* on the system. However locks can make the system very slow [44], and a service that allows for concurrent overlapping operations is desirable.

[5]Atomicity is also referred to as *linearizability* [44] or strong consistency.

tion of an instantaneous read- write memory. Note that the executions of Fig. 4b do not satisfy this property, so they are not atomic. Implementations of atomic shared memory is important because it ensures modular application design. For instance, an application (such as a bank account or a reservation system) can be designed assuming the existence of instantaneous shared variables; using atomic variables in place of instantaneous variables continues to ensure that the overall application works correctly[6].

1.2.2. Problem Statement and Solutions (Informal)

The goal of (atomic) shared memory emulation is to design protocols of write and read clients and servers so that the overall execution is always atomic, and every read and write operations eventually completes so long as the number of server failures is no bigger than f[7]. There are several solutions to the shared memory emulation problem as it is studied extensively in both theory and engineering research. A well-known solution comes from [8]; a reader interested in appreciating systems engineering aspects is also encouraged to read [24]. While [8, 24] use replication for fault tolerance, there is a rich literature in developing provably consistent erasure coding based shared memory emulation algorithms (a non-exhaustive list is ([1, 42, 27, 15, 16, 25, 52]), and also some system implementations [20]. For the purposes of the discussion here, we simply provide a high level understanding of the structure of typical shared memory emulation algorithms.

In shared memory emulation algorithms (e.g., [8, 35]), write operations send the new value to N server nodes and wait until getting acknowledgements from c_W servers before completing the operation. Similarly, a read operation sends a read request to all N servers and waits for responses from at least c_R nodes in the system before reading/decoding the value. To tolerate f server failures—that is, to ensure that write and read operations complete even if f servers fail—it is required that $c_W, c_R \leq N - f$. For every pair of complete write and read operations, there are at least $c_R + c_W - N$ servers that received the value of the write operation, and responded to the read operation. Thus, for repetition based schemes [8, 24], where each server simply stores the latest version it receives in an uncoded form, choosing c_R, c_W so that $c_R + c_W > N$, suffices to ensure that every read operation that begins after a write operation completes will "see" the value of the write, or see a later value. Indeed, in such algorithms, if $c_R + c_W > N$, the replication-based protocols ensure atomic consistency. A natural extension of this principle indicates that when a maximum distance separable (MDS) code is used [42, 27, 16, 25], the dimension of the erasure code is chosen to be $c_R + c_W - N$. Thus, based on the structure of prior protocols, the requirement of ensuring consistency can be expressed as follows[8]:

[6]In practice, systems are sometimes designed with weaker requirements as compared with atomicity for the sake of better performance, i.e., faster read/write operations, see for e.g., [83, 11, 63],

[7]This property is known as the *non-blocking* or *wait-freedom*, property, since it also implies that overlapping operations must be able to proceed without blocking each other.

[8]The reader may note from Fig. 4a that ensuring consistency is more complex and involved than assumption **A4**, and requires careful protocol design; however, for the sake of the discussion here, we simply use **A4** as a proxy for the desired consistency criteria. Also, property **A4** assumes that there is some ordering of the writes/versions; protocols such as [8, 35] also take steps to ensure a non-unique ordering of the write operations.

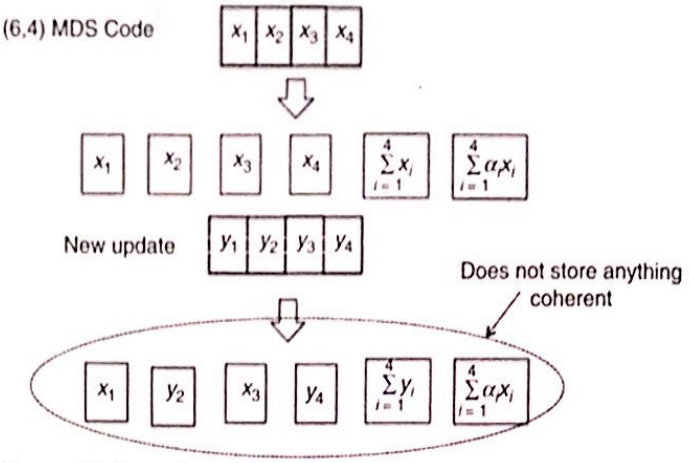

Figure 5. Use of erasure coding for shared memory emulation. The figure demonstrates why a server which receives a new updated codeword symbol cannot simply replace the stored codeword symbol.

A4—Consistency: The latest among all the versions that have been propagated to at least c_W servers, or a later version, must be decodable from any set of c_R servers.

Erasure coding presents interesting algorithmic and coding challenges in asynchronous distributed systems where consistency is important. This is because, when erasure coding is used, no single server stores the data in its entirety; for instance, if a maximum distance separable (MDS) code of dimension k is used, each server only stores a fraction of $1/k$ of the entire value. Therefore, for the read client (decoder) to decode some version of the data, at least k servers must send the codeword symbols corresponding to the same version. In particular, when a write operation updates the data, a server cannot delete the old version before ensuring that the new version has propagated to a sufficient number of servers. As a consequence, servers cannot simply store the latest version they receive; they have to store older versions at least until a sufficient number of codeword symbols corresponding to the newer version has been propagated (See Fig. 5). In fact, in situations where there are multiple concurrent writes, servers may have to store multiple codeword symbols, one corresponding to each concurrent operation. Indeed, the main algorithmic challenge of erasure coding based shared memory emulation algorithms developed previously, is to determine in a decentralized setting that a new version has propagated to a sufficient number of servers before deleting older versions [42, 27, 15, 16, 25, 53, 52, 15, 2, 1, 36]. Importantly for our purposes, the fact that servers have to store multiple versions can offset the storage cost gains of erasure coding, especially when the degree of concurrency is high. Next we summarize *multi-version codes* [87, 88, 89], which study the storage cost of shared memory emulation from an information-theoretic perspective.

1.3. A Coding Framework Related to Shared Memory Emulation

We here describe a coding framework called multi-version coding that simplifies the shared memory emulation problem and yet keeps its essential aspects that pertain to its storage cost. Specifically, in addition to fault tolerance, the multi-version

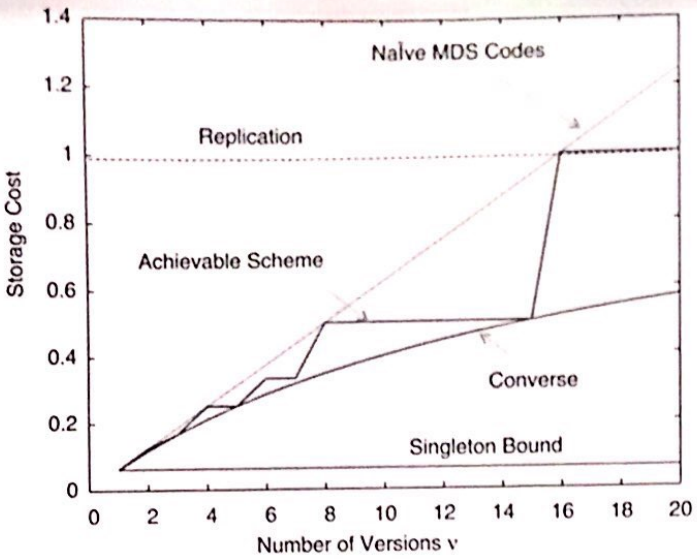

Figure 6. Summary of results of [89].

coding framework incorporates arbitrary asynchrony (**A1**), decentralized nature (**A3**) and consistency (**A4**) as discussed in Sections 1.2.1, 1.2.2. We will discuss modeling assumption (**A2**) in Section 1.4. The multi-version coding problem is parametrized by positive integers n, c_W, c_R, v. There are n servers, and there is a message/variable with v versions W_1, W_2,..., W_v. The versions are assumed to be totally ordered as $W_1 \prec W_2 \prec \ldots \prec W_v$, with a higher ordered message version being interpreted as a more recent version as compared to a lower ordered version. We model asynchrony (**A1**) as follows: Every server receives an arbitrary subset of the versions (at one shot) and encodes the received versions. More formally, let $\mathcal{S}_i \subseteq \{1,2,...,v\}$ be the set of versions received by the ith server. Let $\mathcal{S} = (\mathcal{S}_1,...,\mathcal{S}_n)$; note that $\mathcal{S}$ is an element in the power set of $\{1, 2,..., v\}^n$. We refer to $\mathcal{S}_i$ the *state* of server i, and $\mathcal{S}$ as the global *state of the system*. Denoting $S_i = \{s_1, s_2, ..., s_m\}$ the i-th server stores a codeword symbol generated by an encoding function $\varphi^{(i)}_{\mathcal{S}}$ that takes an input, $(W_{s_1}, W_{s_2},..., W_{s_m})$, and outputs an element in Q, where Q is the alphabet of coding. Note that the decentralized nature (**A3**) of the system is implicit in the model since server i's encoding function $\varphi^{(i)}_{\mathcal{S}}$ depends on $\mathcal{S}_i$ alone, and does not depend on the the global state vector $\mathcal{S}$.

Decoding Constraint: For any state $\mathcal{S} = (\mathcal{S}_1,...,\mathcal{S}_n)$ we refer to a message version that has propagated to at least c_W servers as a *complete* version. The server encoding functions $\varphi^{(i)}_{\mathcal{S}_i}$, $i = 1,2,...,n$ have to be designed such that, the decoder must be able to recover from any c_R servers, the latest c omplete version as per the ordering $\prec$, or a later version. That is, a decoder must be able to decode, from any c_W servers, W_ℓ where $\ell \geq \max\{j : |\{i : j \in \mathcal{S}_i\}| \geq c_W\}$. The decoding constraint has been defined based on consistency requirements (**A4**).

Achievable schemes: With replication, the storage cost per server is the size of one version. Now, suppose we aim to use an MDS code. For a decoder that connects to c_R servers, a complete version is present at least $c_W + c_R - n$ of those servers. So using a dimension of $c_W + c_R - n$ suffices for the decoder to decode the complete version. Furthermore, note that storing simply the latest version at a server does not suffice, since from a server's perspective, it is not aware of which version is complete. Therefore, a server has to store a codeword symbol of a code of dimension $c_W + c_R - n$ for each version it receives. In the worst case, the storage cost per server is $(v/(c_W + c_R - n))$. In our preliminary work, we have improved upon the best of replication and erasure coding by varying the dimension of the code used for the v versions. Our results are depicted in Fig. 6. To appreciate the jagged nature of the achievable scheme, it can be verified that each server storing simply the latest version with dimension $\lceil \frac{v}{v(c_W+c_R-n)+1} \rceil$ suffices. The converse depicted in Fig. 6 carries an instructive message: *in distributed storage systems, in addition to fault tolerance, there is an inevitable price to be paid in terms of redundancy overhead to maintain consistency.* The converse derivation is combinatorial in nature, discovering the worst-case states that force a lower bound on the storage cost.

1.3.1. Where Do Classical Distributed Storage Erasure Codes Fit?

Classical erasure codes may be derived as a special case of multi-version codes for $v = 1$. From a modeling perspective, this translates to the system being synchronous, or completely centralized. For instance, if the system is synchronous, then all non-failed servers receive each version completely. Then storing simply the latest version with dimension c_R code suffices for the decoder. Even if the system is asynchronous, i.e., a server receives an arbitrary subset of the v versions in the system, a centralized system where each server knows the global system state leads to the classical erasure coding framework. To see this, observe that with global system state information, each server is aware of the the latest complete version. Therefore, a server with the latest complete version simply encodes it using a code of dimension $c_W + c_R - N$ and stores the corresponding codeword symbol; a server that does not posses the latest common version does not store any information. These examples show that new coding ideas are required only when the model accounts for both the asynchronous nature and decentralized nature of the system.

1.4. Extensions and Open Problems

Classical erasure codes for distributed storage may be interpreted as an optimistic view of the system: it assumes that the system is synchronous, and that nodes have instantaneous and global system state information. In contrast, the multi-version coding framework takes a pessimistic/conservative view of storage systems (Fig. 7). Specifically the model assumes that the system is completely asynchronous, i.e., every version arrival pattern is possible; it also assumes that nodes do not have even stale or partial information of system state. However, in practice, we expect the system to operate somewhere in between these two extremes, specifically nodes may be able to opportunistically obtain stale or partial information of the system. This opens the door to many important open questions that enable code constructions.

- The multi-version coding is a worst-case formulation, where the decoding requirement as well as the storage cost characterization applies to every possible state. An open question is the potential storage cost reductions that may be obtained by studying the average storage cost, and/or allowing for a small probability of decoding error after invoking an appropriate probability measure on the state space.

- In several settings, either because of the topology or the internal gossip messages exchanged among the servers, servers obtain local and possibly stale side information (e.g., a server may know which server has some of the older versions.) In the extreme case where server has global and instantaneous side information, the system reverts back to the classical setting as explained in Section 1.3.1. The question of how to design codes that exploit stale or local side information about the states, and a quantitative understanding of the potential storage cost gains is a promising area of future work.

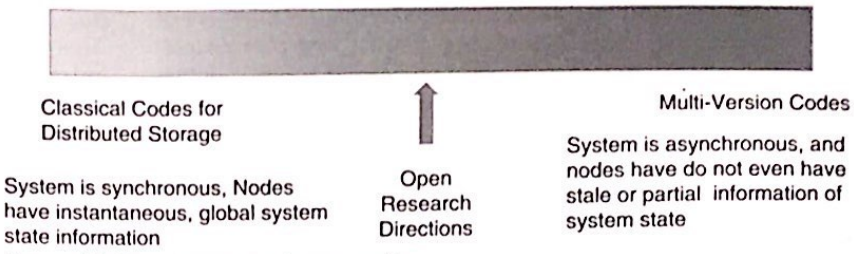

Figure 7. Open areas of research.

- The multi-version coding setting assumes that the different versions are independent. In some applications, subsequent writes and reads tend to have correlations that can be exploited to reduce the memory overhead in such applications. Our preliminary work [5] explores this direction and derives achievable schemes and characterizes their storage cost. However, questions related to optimality and practical low complexity code constructions remain open.

The second direction of open problems comes from understanding and developing connections between distributed computing systems and multi-version codes. We describe some of them briefly here.

- In our recent work [17], we have proved formal information theoretic lower bounds on the shared memory emulation problem inspired by converse results for the multi-version coding framework. Specifically, the result of [17] shows that the multi-version coding converse applies to the general shared memory emulation problem when $v = 2$, and for non-interactive protocols when $v > 2$. It is not known whether interactive protocols can improve upon the storage cost of multi-version coding—the reason for this gap in understanding is that multi-version coding does not capture assumption **A2** in Section 1.2.1 very well. While studying this question in the context of the general shared memory emulation problem might be challenging, a possible starting point is by making the toy model of [89] interactive.

- The shared memory emulation problem is a special case of the general *replicated state machine* [60, 9] problem in distributed computing, where the goal is to emulate an arbitrary state machine in a distributed asynchronous system in a fault tolerant manner. The replicated state machine problem is much more complicated than the shared memory emulation problem, and is an object of extensive study in distributed systems literature. Like shared memory emulation, solutions to the replicated state machine problem—for specific state machines—form the basis of several cloud based services (e.g., Google Spanner [21]). However, the question of how to encode state machines is not yet well studied by either the distributed systems or the coding theory communities. A good starting point is in reference [12], which generalizes basic coding theoretic concepts such as the Hamming distance to replicated state machines.

In addition to the above mentioned extension and open problems, a deeper understanding of the system from a networking and resource allocation viewpoint, which can guide an engineer on how to choose coding parameters such as the length, dimension, and the extent of history to be stored, based on the total storage budget, frequency of updates, the degree of asynchrony, and other system level parameters is a broadly open and important area of research.

2. Part II: Coding for Distributed Computing Using Unreliable Components: Errors, Faults, Stragglers

The roots of computing using unreliable components go as far back as work of von Neumann in 1956 to a work [84]. He considers a problem where each gate is unreliable, and provides replication-type strategies with repeated majority decoding make the overall computation reliable. von Neumann's work was inspired both by Shannon's 1948 paper [72] and by his interest in understanding biological computation[9], e.g., in our nervous system[10]. von Neumann's work was followed up in a sequence of works, including those of Pippenger [66, 67], Taylor [77] (see also [78]), Hadjicostis [37], and Hajek and Weller [40] who provided achievable strategies for specific operations. Taylor's work started the area of decoding using unreliable circuits, focusing on low-density parity-check codes of Gallager, and allowing for errors in his iterative decoding algorithm. This work has been followed up extensively recently, e.g. Varshney [80], Vasic et al. [81, 82], Dolecek et al. [94, 46], with an interesting recent work of Vasic showing that low-error rates in the decoding implementation can help *improve* the performance of decoding. Pippenger's work [66] not only provided achievable strategies for linear computations, it also utilized work of Dobrushin and Ortyukov [26] to lay down fundamental limits by tightening the data-processing inequality in the context of scalar random variables. This direction on fundamental limits was extended and explored in the works of Evans and Schulman [33], and Erkip and Cover [32] on "quantified/strong" data-processing inequalities, that has led to recent resurgence in this direction, with broadening of problems to "information-dissipation" [68, 18, 6, 69].

In a deeply related but separate body of work, inspired by distributed and parallel computing systems that have been implemented in the last few decades, there has been significant work on addressing computing under "processor" unreliability, where a processor is assumed to be constituted by a large number of gates and attached memories.

[9]Shannon himself was examining the noisy computing problem at the same time, focusing on unreliable relays [62].

[10]It is now widely believed that the brain indeed uses error-correcting codes [73, 64] (his paper is titled "Probabilistic Logics and the Synthesis of Reliable *Organisms* from Unreliable Components"). Whether it does so to compute efficiently is far from established, although the thought behind Barlow's famous and controversial "Efficient Coding Hypothesis" does suggest so [13].

Addressing processor unreliability is even today thought to be "one of top 10 challenges for exascale computing" [59] (i.e., larger-scale supercomputing, where processors can yield undetected errors). Coding-theoretic work in this direction was initiated by Huang and Abraham [47], who named it "Algorithm-Based Fault Tolerance" (ABFT), which is today a thriving research area with hundreds of papers (e.g. [43, 49, 37, 50]). These works provide techniques of error-correction for large-scale computing systems, including modern supercomputing systems. A closely related application area is distributed/cloud computing systems, where unreliability manifests itself in the form slow processing nodes (called *stragglers*), which significantly slow down the entire computation [23]. Use of coding techniques in cloud computing systems to address straggler bottlenecks was pioneered in the recent work of Lee et al. [56]. In [56], the authors demonstrated the power of ABFT-like coding techniques for linear computations via novel expected time-analysis based on exponential-tail models of processing times, in addition to experimental results. Interestingly, the coding techniques developed in the ABFT literature turn out to be suboptimal in the sense of error/erasure-tolerance for operations such as matrix-vector and matrix-matrix multiplication. Recent work in information theory has advanced on these constructions while obtaining, in some cases, fundamental limits as well. For instance, for the problem of matrix-matrix multiplication (Section 2.2), the work of Yu, Avestimehr, and Maddah-Ali [95] provides a coded computing construction that they call "Polynomial coding," that provides scaling-sense improvements on ABFT by comparing ABFT's performance with fundamental limits. This was followed by our own recent work [34] which provides scaling sense improvements on Polynomial codes. The information-theoretic approach has much to offer.

More recently, coded computing results have been obtained for convolutions [91, 31, 74, 95], solving linear inverse problems and PageRank [92], distributed gradient descent [30, 75, 76, 70, 41], linear regression and classification [56, 30], logistic regression [90], distributed iterative optimization [51, 10], and even separable non-linear functions [57], etc. For any computation that can be split into tasks, optimized approaches have been proposed by Joshi, Wornell, and Soljanin, and collaborators [86, 48, 85, 3, 4] for straggling processors, which focus on problems of task allocation using rigorous queuing-theoretic models. Coding techniques have also been used to reduce communication requirements even when processing nodes are completely reliable, and a notable work here is that on "Coded MapReduce" by Li, Avestimehr, and Maddah-Ali [58].

In the following, we will discuss the development of the field from the perspective of three key problems. In Section 2.1, we will discuss matrix-vector multiplication under processor level errors or stragglers [47, 56, 30, 75, 76, 70, 41] as well as gate-level errors [66, 93]. In Section 2.2, we will discuss advances in matrix-matrix multiplication, including recent works [47, 95, 34]. Finally, in Section 2.3, we will end with a recent coded-computing construction of Dutta et al. [29] that connects back to the inspiration of von Neumann's seminal work: the brain. This construction provides reliability to large networks of the nonlinear McCulloch-Pitts neuron models that are revolutionizing computing and inference systems today.

2.1. Coded Matrix-Vector Multiplication

The problem of computing linear transforms of high-dimensional vectors is *the* critical step [22] in several machine learning and signal processing applications. Dimensionality reduction techniques such as Principal Component Analysis (PCA), Linear Discriminant Analysis (LDA), taking random projections, etc., require the computation of short and fat linear transforms on high-dimensional data. Linear transforms are the building blocks of solutions to various machine learning problems, e.g., regression and classification etc., and are also used in acquiring and pre-processing the data through, e.g., filtering. Fast and reliable computation of linear transforms are thus a necessity for low-latency inference [22].

Large processing nodes: The goal is to compute the product $\mathbf{Ay}$ of the matrix $\mathbf{A}$ with the vector $\mathbf{y}$. We first discuss the case with "large" processing nodes, where e.g., the memory-size of each processing node can scale with the problem-size. The first strategy here was proposed in the seminal work of Huang and Abraham work on ABFT [47]. The idea of using coding to mitigate the *straggler effect* in distributed systems was pioneered by [56], which used what happens to be a special case of ABFT to speed up matrix-vector products. The authors in [56] show, both analytically (using models inspired from distributed systems) and experimentally (on Amazon's EC2 cluster), that significant speed-ups in expected overall computation time can be obtained by using these techniques. The core idea of [56] is to code each column of the matrix $\mathbf{A}$ by multiplying it with the same generator matrix of a linear code, resulting in a coded matrix $\mathbf{A}_{\text{coded}}$. Now, we distribute the rows of the matrix $\mathbf{A}_{\text{coded}}$ among different processing nodes, and each node computes its own matrix-vector product. Because linear combinations of codewords of a linear code are also codewords, the resulting concatenation of outputs is also a codeword. Hence, the computation can proceed without waiting for a few straggling nodes by using erasure decoding to fill in for their outputs.

What if, due to communication or memory bottlenecks, each node cannot even compute a whole dot product of a row of of an $M \times N$ matrix $\mathbf{A}$ with the $N \times 1$ vector $\mathbf{y}$? Recently, our work [30] as well as Tandon et al. [75] (motivated by a different application) simultaneously arrived at strategies for coding matrix-vector multiplication by encoding the matrix into a sparse coded matrix. These codes, that we call "Short-Dot" codes because they distribute large matrix-vector products to processing nodes that compute short dot products, are illustrated below through an example.

Consider a 2×4 *matrix* $\mathbf{A} = [\mathbf{a}_1^T\ \mathbf{a}_2^T]^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{14} \end{bmatrix}$ *and a* 4×1 *vector* $\mathbf{y} = (y_1, y_2, y_3, y_4)^T$. *Can we compute* $\mathbf{Ay}$ *over* $n = 4$ *nodes such that,* (i) *each node uses a shortened version of* $\mathbf{y}$, *i.e., at most three of four scalars* y_1, y_2, y_3, y_4, *and* (ii) *the overall computation is tolerant to one straggler,* i.e., 2 *of the three nodes suffice to recover* $\mathbf{Ay}$? *Short-Dot codes use the following strategy: Node* i *computes* $(\mathbf{a}_1 + \mathbf{a}_2 i + \mathbf{z} i^2)\mathbf{y}$, $i = 1, 2, 3, 4$ *so that from any* 3 *of the* 4 *nodes, the polynomial* $p(x) = (\mathbf{a}_1\mathbf{y} + \mathbf{a}_2\mathbf{y}x + \mathbf{z}\mathbf{y}x^2)$ *can be interpolated. Vector* $\mathbf{z} = (z_1\ z_2,\ z_3,\ z_4)$ *is chosen to satisfy* $\mathbf{a}_{1i} + i\mathbf{a}_{2i} + i^2 z_i = 0$ *for* $i = 1, 2, 3, 4$, *so that node* i *does not require* y_i.

To understand the performance of Short-Dot from a quantitative perspective, consider a setting shown in Fig. 8 where each processing/worker node stores[11] MN/m linear combinations of the entries of $\mathbf{A}$ and N/n non-zero entries of $\mathbf{y}$. Assume that P processing nodes perform the computation, we use "recovery threshold" $K(m, n)$—the minimum number of processing nodes required by a fusion node to recover the matrix-vector product $\mathbf{Ay}$—to measure

[11] Although we use the word "store" in our descriptions, note that the restrictions on the amount of $\mathbf{A}$ and $\mathbf{x}$ need not be only due to memory overhead; these restrictions may be motivated by communication costs as well.

its resilience. Note that the maximum number of straggling nodes that a scheme can tolerate is $P - K(m, n)$. $K(1,1)$ is clearly 1, and translates to a replication strategy. In the MDS code based strategy of [56, 47], $\mathbf{A}$ is split as $\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_m \end{bmatrix}$ and node $i \in \{1, 2, \ldots, P\}$ computes $(\Sigma_{j=1}^{M} g_{ij}\mathbf{A}_j)\mathbf{y}$, where $\mathbf{G} = (g_{ij})$ forms the generator matrix of a (P, m) MDS code[12] The fusion node can recover $\mathbf{Ay}$ from any m nodes, which gives $K(m, 1) = m$. Interestingly, going beyond $n = 1$ – that is, using the full vector $\mathbf{y}$ at all nodes—requires interesting coding strategies as demonstrated by the above example. Short-Dot obtains a recovery threshold of $K(m,n) = (m/n) + P(1-(1/n))$; via a converse result, this recovery threshold is approximately optimal [28, 30].

In fact, the number of non-zero entries in the stored, coded submatrix at each processing node is MN/mn since every row of the coded matrix only has N/n non-zero entries at *pre-defined* locations. Thus it requires even smaller memory than MN/m. The key difference between Short-Dot and MDS codes is that Short-Dot allows the vector y to be shorter than its full-length N, thus allowing for cheaper communication of parts of y to each processing node and computation of shorter dot-products at each processing node as compared to MDS code.

Concurrent work [75] also discovered *gradient coding*, which incorporates the essential coding idea of Short-Dot in the context of a single dot product in an important application: distributed gradient descent. In an iteration of distributed gradient descent over data $\mathbf{D} = (\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_P)$ where $\mathbf{D}_i$ is the data stored in the i-th processing node, the gradient update step requires each processing node to to compute a gradient $g(\mathbf{D}_i)$, which is used by a master node to compute $\Sigma_{i=1}^{P} g(\mathbf{D}_i)$. In [75], Tandon et al. view the operation of a master node as a single *dot-product* of the vector [1 1 ... 1] with the vector $[g(\mathbf{D}_1)\ g(\mathbf{D}_2)\ \ldots\ g(\mathbf{D}_P)]$. They use a coding technique similar to Short-Dot to complete the iterations in presence of straggling worker nodes. They also introduce the notion of *partial stragglers*, [75] which opens up new research directions by modeling the fact that stragglers—unlike faulty nodes—eventually complete their operation and their outputs can be used in the eventual computation.

A direct application of matrix-vector multiplication is solving sparse linear inverse problems using iterative matrix-vector products, such as for finding eigenvectors of a matrix through, e.g., the PageRank algorithm. What is new in these problems is that the answer slowly converges to the true solution. The stragglers thus simply have fewer iterations compared to fast nodes, but their outputs are still useful. In [92], we use a decoding algorithm inspired by weighted least-squares to fully utilize the results from both fast nodes and stragglers by assigning weights to different nodes based on their proximity to convergence. Compared to erasure-coding based coded computing, this coding method achieves graceful degradation of remaining error with increasing number of stragglers.

[12]In fact, aggregating these coded matrices forms an MDS coded matrix, $\mathbf{A}_{\text{coded}}$. However, we use the term $\mathbf{A}_{\text{coded}}$ more generally to denote a coded version of the matrix A for an arbitrary linear code, not necessarily an MDS code.

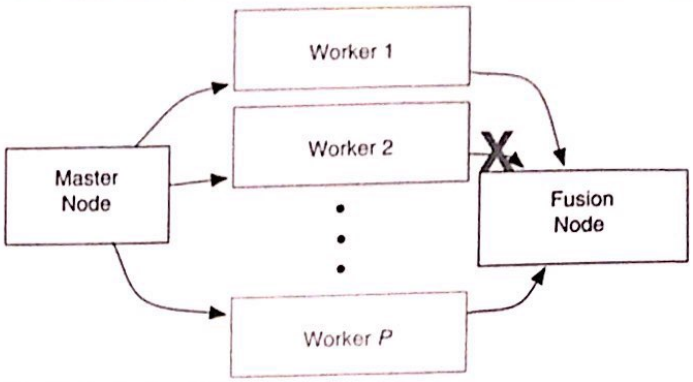

Figure 8. System model.

Both theoretical and experimental results show substantial advantages of the proposed algorithm over replication schemes[13]. Surprisingly, if the ordinary matrix-inverse algorithm is used for decoding, the remaining error shoots up because MDS coding matrices appear to be ill-conditioned, which is also recognized in the work of Haikin and Zamir [39, 38] on analog coding for erasure channels. By carefully combining the (partial) results of the straggling nodes (instead of ignoring them), we are able to circumvent the difficulty of ill-conditioned matrix inverse and achieve orders of magnitude reduction of remaining error in experiments.

Small processing nodes (i.e., single gates): When considering models with gate-level errors, the cost of error-detection can be significant, so it is important for the models to consider errors. Further, the implementation has to be fully decentralized: decoding itself can suffer from errors.

For this problem, the fundamental limits of "strong" data-processing inequality provide an important intuition: along any single "path" in the circuit, errors will accumulate, and information will dissipate. This seems to present a pessimistic picture. Is there any hope for reliable computation in the Shannon sense [72]? First, note that the overall computation error-probability cannot be lower than the error-probability of the last gate, so the best we can hope for is that the overall error probability is close to the last gate's error probability. For the specific problems of binary matrix-vector multiplication, our recent work [93] shows that even with all gates noisy, this is achievable, and further, that sophisticated error-correction techniques that our community has developed offer scaling-sense advantages.

To describe our technique, it is useful to first discuss a fundamental limit, namely, Lemma 2, in the work of Evans and Schulman [33]. They derive an upper bound on mutual information between a binary input and the values carried by a set of wires by accumulating mutual information across those wires. This suggests an achievability: if the number of paths that a circuit can keep accumulating information from keeps increasing as the information dissipates along each individual path, it may

[13]It is interesting that the diversity gain achieved by *the replication strategy* here is smaller than that can be achieved in communications over random fading channels, because the results at stragglers are deterministic functions of those at fast nodes for the same inverse problem. Fundamental limit on obtained diversity gains are worthy of further investigation.

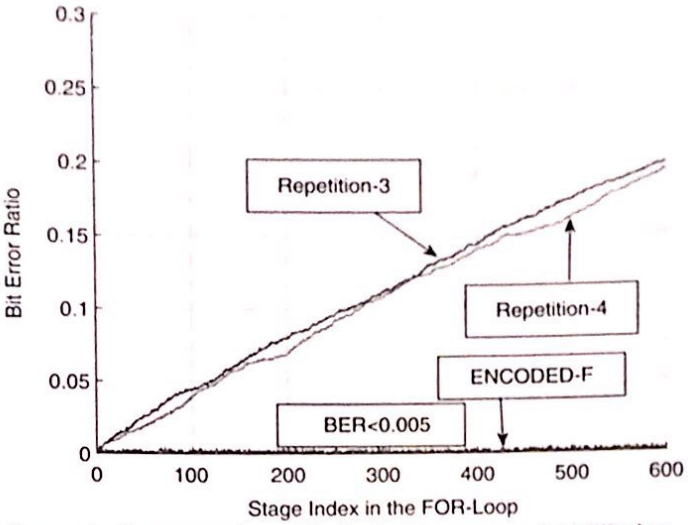

Figure 9. A comparison of attained error-probability by our technique ENCODED with reptition/replication- type approaches. ENCODED is able to keep errors bounded even as the computation proceeds.

be possible to offset the losses with the gains, and keep errors suppressed.

Our results [93] show that this is indeed attainable for computing a binary linear transform $\mathbf{Ay}$. To do so, we encode $\mathbf{A}$, obtaining $\mathbf{A}_{\text{coded}}$ with coded columns, that is, each column is a codeword. The coding technique itself is a carefully chosen LDPC code. Now, instead of performing distributed dot-products of individual rows of $\mathbf{A}_{\text{coded}}$ with $\mathbf{y}$, we perform scalar-vector multiplications of y_i, the i-th element of $\mathbf{y}$, with the i-th column of $\mathbf{A}_{\text{coded}}$. Note that the resulting scalar-vector product is a codeword itself. The key step is this next one: we add these codeword vectors across different indices i in a tree architecture, and at every intermediate node in the tree, we embed a noisy decoder (utilizing results from [94, 46]). These decoders enable repeated error-suppression as the computation advances (see Fig. 9), and the accumulation of information from various paths enables the computation to compensate for information dissipation. The resulting error probability is shown, through theoretical results and simulations, to remain bounded by a constant throughout the computation. Because the repeated error-suppression using embedded decoders is a critical aspect of our strategy, we call it "ENcoded COmputation with Decoders EmbeddeD," or "ENCODED").

2.2. Coded Matrix Multiplication

In this section, we focus on the problem of multiplying two $N \times N$ matrices[14] $\mathbf{A}$, $\mathbf{B}$. Consider the setting where each processing node stores N^2/m linear combinations of the entries of $\mathbf{A}$ and N^2/n linear combinations of the entries of $\mathbf{B}$. Assume that P processing nodes perform the computation, we evaluate the straggler tolerance of a technique by its recovery threshold $K(m, n)$ of the technique. As before, $K(1,1)$ is clearly 1, and translates to a repetition based strategy.

[14]We assume that both matrices are square for the sake of simplicity. The results of this section apply for multiplication of matrices of arbitrary dimension for mild assumptions on the matrix dimensions.

	Worker (:, 1)	Worker (:, 2)	Worker (:, 3)
Worker (1, :)	$\mathbf{A}_1$	$\mathbf{B}_1$	$\mathbf{B}_2$	$\mathbf{B}_1 + \mathbf{B}_2$
Worker (2, :)	$\mathbf{A}_2$			
Worker (3, :)	$\mathbf{A}_1 + \mathbf{A}_2$			

Figure 10. ABFT matrix multiplication [47] for $P = 9$ processing nodes with $m = n = 2$, where $\mathbf{A} = \begin{bmatrix}\mathbf{A}_1 \\ \mathbf{A}_2\end{bmatrix}\mathbf{B} = [\mathbf{B}_1 \ \mathbf{B}_2]$. The recovery threshold is 6.

A natural generalization of the MDS code based strategy, which we call *one-dimensional MDS coding*, where $\mathbf{A}$ is encoded using and MDS code gives $K(m, 1) = m$; similarly, encoding $\mathbf{B}$ gives $K(1, n) = n$. We review here three strategies (i) ABFT matrix multiplication [47] (also called *product-coded matrices* in [55]), (ii) Polynomial codes [95] and (iii) MatDot codes [34] each with successively improving, i.e., smaller, recovery threshold. Rather than go into the technical details, we give three examples for the case where $m = n = 2$—i.e., each processing node stores half of $\mathbf{A}$ and half of $\mathbf{B}$—that convey the ideas used. We begin by describing ABFT matrix multiplication.

Example (ABFT Codes [47], Fig. 10, recovery-threshold = 6) *Consider two $N \times N$ matrices*

$$\mathbf{A} = \begin{bmatrix}\mathbf{A}_1 \\ \mathbf{A}_2\end{bmatrix}, \mathbf{B} = [\mathbf{B}_1 \ \mathbf{B}_2],$$

Can we compute $\mathbf{AB}$ over P nodes such that, **(i)** *each node uses one linear combination of $\mathbf{A}$ and one linear combination of $\mathbf{B}$ and* **(ii)** *the overall computation is tolerant to $P - 6$ stragglers, i.e., 6 nodes suffice to recover $\mathbf{AB}$? ABFT codes use the strategy as per Fig. 10, where 4 of the 9 worker nodes compute $\mathbf{A}_i, \mathbf{B}_j$, i, j in $\{1, 2\}$ and the remaining worker nodes compute $\mathbf{A}_i(\mathbf{B}_1 + \mathbf{B}_2)$, $(\mathbf{A}_1 + \mathbf{A}_2)\mathbf{B}_i$, $(\mathbf{A}_1 + \mathbf{A}_2)(\mathbf{B}_1 + \mathbf{B}_2)$ for $i = 1, 2$ respectively. The general principle of ABFT is to encode the rows of $\mathbf{A}$ and the columns of $\mathbf{B}$ using systematic MDS codes of dimension m, n respectively.*

The question is whether the recovery threshold of 6 is optimal was addressed in [95], which gave the following, elegant, polynomial code construction.

Example (Polynomial Codes [95], Fig. 11, recovery-threshold = 4) *Consider two $N \times N$ matrices*

$$\mathbf{A} = \begin{bmatrix}\mathbf{A}_1 \\ \mathbf{A}_2\end{bmatrix}, \mathbf{B} = [\mathbf{B}_1 \ \mathbf{B}_2],$$

can we compute $\mathbf{AB}$ over P nodes such that, **(i)** *each node uses one linear combination of $\mathbf{A}$ and one linear combination of $\mathbf{B}$ and* **(ii)** *the overall computation is tolerant to $P - 4$ straggler, i.e., 4 nodes suffice to recover $\mathbf{AB}$? Polynomial codes use the following strategy: Node i computes $(\mathbf{A}_1 + \mathbf{A}_2 i)(\mathbf{B}_1 + \mathbf{B}_2 i^2)$, $i = 1, 2, \ldots P$, so that from any 4 of the P nodes, the polynomial $p(x) = (\mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_1 x + \mathbf{A}_1\mathbf{B}_2 x^2 + \mathbf{A}_1\mathbf{B}_2 x^3)$ can be interpolated. Having interpolated the polynomial, the coefficient (matrices) can be used*

to evaluate $\mathbf{AB}$ as $\begin{bmatrix}\mathbf{A}_1\mathbf{B}_1 & \mathbf{A}_1\mathbf{B}_2 \\ \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_2\mathbf{B}_2\end{bmatrix}$.

In [34], we improve upon the recovery threshold of polynomial codes through a construction called *MatDot* codes. Unlike ABFT

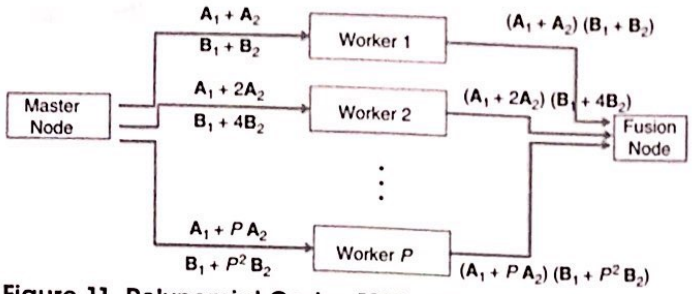

Figure 11. Polynomial Codes [95] with $m = n = 2$ where $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}, B = [B_1 \ B_2]$. The recovery threshold is 4.

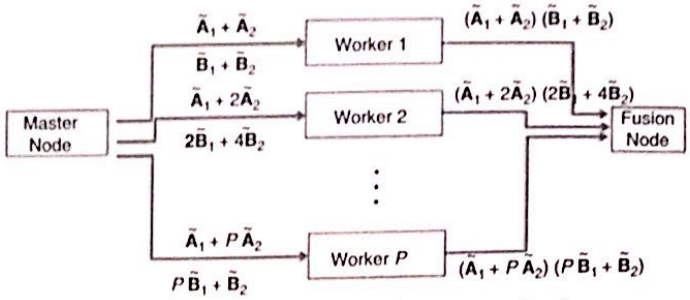

Figure 12. Matdot Codes [34] with $m = n = 2$ where $A = [\tilde{A}_1 \ \tilde{A}_2], B = \begin{bmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{bmatrix}$. The recovery threshold is 3.

and polynomial codes, MatDot splits the matrix **A** column-wise and matrix **B** row-wise.

Example (MatDot Codes [34], Fig. 12, recovery-threshold = 3.) *Consider two $N \times N$ matrices*

$$A = [\tilde{A}_1 \ \tilde{A}_2], B = \begin{bmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{bmatrix},$$

can we compute **AB** *over P nodes such that,* **(i)** *each node uses one linear combination of* **A** *and one linear combination of* **B** *and* **(ii)** *the overall computation is tolerant to $P - 3$ straggler, i.e., 3 nodes suffice to recover* **AB**? *MatDot codes use the following strategy: Node i computes $(\tilde{A}_1 + \tilde{A}_2 i)(\tilde{B}_1 i + \tilde{B}_2)$, $i = 1, 2, \ldots P$, so that from any 4 of the P nodes, the polynomial $p(x) = \tilde{A}_1\tilde{B}_2 + (\tilde{A}_1\tilde{B}_1 + \tilde{A}_2\tilde{B}_2)x + \tilde{A}_2\tilde{B}_1x^2$ can be interpolated. Having interpolated the polynomial, the product* **AB** *is simply the coefficient of x, that is* $\mathbf{AB} = \tilde{A}_1\tilde{B}_1 + \tilde{A}_2\tilde{B}_2$

In fact, if $m = n = o(\sqrt{P})$ then the recovery threshold of one dimensional MDS, ABFT matrix multiplication, polynomial codes and MatDot codes is respectively $\Theta(P), \Theta((m-1)\sqrt{P}), \Theta(m^2), \Theta(m)$. There are differences between the schemes in terms of node communication costs and decoding costs; these and other related aspects are described in [34]. [34] also shows how to apply these ideas for multiplying more than two matrices, and generalizes and unifies Polynomial codes and MatDot codes to "PolyDot" codes that tradeoff between communication complexity and recovery threshold.

2.3. Coded Neural Networks

Deep Neural Networks (DNNs) are being extensively used in many inference applications. A DNN is a multilayer network of artificial neuron models (first developed by McCulloch and Pitts), where each layer performs a linear transform on its input vector from the previous layer. Training of DNNs is extremely time-intensive, and has two main steps: "feed-forward step" and "back-propagation." In the feed-forward step, after computing a linear transform on the vector received from the layer to its left, each layer performs scalar nonlinear computations on each element of the vector that results from this linear transform, and then forwards it to the layer on its right. In the back-propagation step, the layer receives the back-propagated vector from a layer on its right, and computes a linear transform before forwarding it to the layer on its left. The vector received by a layer in the backpropagation step is also used to update the layers' linear transform in preparation for the next iteration. An iteration consists of a feed-forward step and a back-propagation step—including the linear transformation update—executed by all layers, where the first layer receives a data point, and the last layer receives an evaluation of a loss function, as their inputs. In this description, for the sake of simplicity of exposition, we assumed that each iteration is for a single data point.

The most computationally intensive step in the training and testing of a DNN is a matrix-vector multiplication. However, a naive extension of ABFT techniques of coded matrix-vector multiplication (e.g. [47, 56]) would require us to encode matrices at every training iteration. This is because these matrices are updated at every iteration. This overhead of encoding can be enormous, and comparable to the cost of the computation itself, which is undesirable. However, our strategy—that we call *CodeNet*—is able to show that by carefully weaving coding into the computation of DNN, we are only required to code vectors (which is low complexity), and not matrices, at every iteration. This enables encoding and decoding on the fly. Suppose W is the weight matrix at any layer of the DNN. Then the most computationally intensive operation in the feed-forward stage of the DNN is the matrix-vector product $s = Wx$ where x comes from the previous layer. Similarly in the back-propagation stage, the bottleneck operation is the computation of vector-matrix product $c^T = \delta^T W$ where δ comes from the next layer. Finally, in the update step, the W is updated as $W \leftarrow W + \mu\delta x^T$.

We divide the matrices and vectors into blocks as $W = \begin{bmatrix} W_{0,0} & W_{0,1} \\ W_{1,0} & W_{1,1} \end{bmatrix}, x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ and $\delta^T = [\delta_0^T \ \delta_1^T]$. In CodeNet, at the beginning of training, the matrix W is coded and stored distributedly (by paying a one-time cost) as follows:

$$W_{\text{coded}} = \begin{bmatrix} W_{0,0} & W_{0,1} & \widehat{W}_{0,2} & \widehat{W}_{0,3} \\ W_{1,0} & W_{1,1} & \widehat{W}_{1,2} & \widehat{W}_{1,3} \\ \widehat{W}_{2,0} & \widehat{W}_{2,1} & x & x \\ \widehat{W}_{3,0} & \widehat{W}_{3,1} & x & x \end{bmatrix},$$

where each block is stored in a different processing node, and 'x' is simply denoting that no block exists at that location.

The straggler/error resilience for feed-forward and back-propagation are obtained through standard ABFT-type MDS coding, the difficulty is in maintaining the coding of the weight matrices even as they are updated because encoding the matrix at each iteration is computationally expensive. We now explain the strategy in a bit more detail, first discussing feed-forward and back-propagation steps, and then discussing how the coding is maintained in the update step:

For the feed-forward stage, one only uses the first two columns as follows:

$$\begin{bmatrix} s_0 \\ s_1 \\ \widehat{s_2} \\ \widehat{s_3} \end{bmatrix} = \begin{bmatrix} W_{0,0} & W_{0,1} \\ W_{1,0} & W_{1,1} \\ \widehat{W}_{2,0} & \widehat{W}_{2,1} \\ \widehat{W}_{3,0} & \widehat{W}_{3,1} \end{bmatrix} x = \begin{bmatrix} W_{0,0}x_0 + W_{0,1}x_1 \\ W_{1,0}x_0 + W_{1,1}x_1 \\ \widehat{W}_{2,0}x_0 + \widehat{W}_{2,1}x_1 \\ \widehat{W}_{3,0}x_0 + \widehat{W}_{3,1}x_1 \end{bmatrix}$$

One can use decoding techniques similar to coded matrix-vector multiplication to get back s_0 and s_1 from the 4 coded vectors $\{s_0, s_1, \widehat{s_2}, \widehat{s_3}\}$ under errors or stragglers. Similarly, for the back-propagation stage, one can only use the first 2 rows of the coded matrix as follows:

$$\begin{aligned} [c_0^T \;\; c_1^T \;\; \widehat{c_2^T} \;\; \widehat{c_3^T}] &= \delta^T \begin{bmatrix} W_{0,0} & W_{0,1} & \widehat{W}_{0,2} & \widehat{W}_{0,3} \\ W_{1,0} & W_{1,1} & \widehat{W}_{1,2} & \widehat{W}_{1,3} \end{bmatrix} \\ &= \begin{bmatrix} \delta_0^T W_{0,0} & \delta_0^T W_{0,1} & \delta_0^T \widehat{W}_{0,2} & \delta_0^T \widehat{W}_{0,3} \\ +\delta_1^T W_{1,0} & +\delta_1^T W_{1,1} & +\delta_1^T \widehat{W}_{1,2} & +\delta_1^T \widehat{W}_{1,3} \end{bmatrix} \end{aligned}$$

Now, from the 4 coded vectors $\{c_0^T, c_1^T, \widehat{c_2^T}, \widehat{c_3^T}\}$, one can decode c_0^T and c_1^T under errors and stragglers. This ensures that both forward computation and back-propagation are resilient to errors. Importantly, by weaving coding in this manner, every sub-matrix is able to update itself at each iteration *without the need to encode matrices afresh*:

Suppose, $\widehat{W}_{0,2} = W_{0,0} + W_{0,1}$. For the update step, if the processing node only has two vectors: δ_0^T and coded vector $\hat{x}_2 = x_0 + x_1$, then it can update itself as

$$\widehat{W}_{0,2} + \mu\delta_0\hat{x}_2^T = \underbrace{W_{0,0} + \mu\delta_0 x_0^T}_{\text{Updated } W_{0,0}} + \underbrace{W_{0,1} + \mu\delta_0 x_1^T}_{\text{Updated } W_{0,1}}.$$

The recovery threshold of this strategy can be improved by utilizing more sophisticated codes, and will appear in [29].

Acknowledgements

We would like to thank Kishori Konwar, Haewon Jeong, Sanghamitra Dutta, and Yaoqing Yang for giving insightful comments on this article. We would also like to thank Zhiying Wang, Nancy Lynch, Prakash Narayana Moorthy, Muriel Medard, A. Dimakis, Mohammad Maddah Ali, Salman Avestimehr, Ramy E. Ali, Farzin Haddadpour and Mohammad Fahim for their help during tutorial preparation. We are grateful to the National Science Foundation and the SRC SONIC center for funding our research, which led to this article.

References

[1] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault- scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74, 2005.

[2] Marcos Kawazoe Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 336–345, 2005.

[3] Mehmet Fatih Aktas, Pei Peng, and Emina Soljanin. Effective straggler mitigation: Which clones should attack and when? *arXiv preprint arXiv:1710.00748*, 2017.

[4] Mehmet Fatih Aktas, Pei Peng, and Emina Soljanin. Straggler mitigation by delayed relaunch of tasks. *arXiv preprint arXiv:1710.00414*, 2017.

[5] Ramy E Ali and Viveck R Cadambe. Consistent distributed storage of correlated data updates via multi-version coding. In *Information Theory Workshop (ITW), 2016 IEEE*, pages 176–180, 2016.

[6] Venkat Anantharam, Amin Gohari, Sudeep Kamath, and Chandra Nair. On hypercontractivity and a data processing inequality. In *Information Theory (ISIT), 2014 IEEE International Symposium on*, pages 3022–3026, 2014.

[7] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[8] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[9] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[10] N. Azian-Ruhi, S. Avestimehr, F. Lahouti, and B. Hassibi. Consensus-based distributed computing. In *Information Theory and Applications Workshop*, 2017.

[11] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[12] Bharath Balasubramanian and Vijay K. Garg. Fault tolerance in distributed systems using fused state machines. *Distributed Computing*, 27(4):287–311, 2014.

[13] Horace B Barlow. Possible principles underlying the transformations of sensory messages. *Sensory Communication*, pages 217–234, 1961.

[14] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI*, volume 6, pages 12–12, 2006.

[15] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 115–124, 2006.

[16] Viveck R Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. In *2014 IEEE 13th International Symposium on Network Computing and Applications (NCA)*, pages 253–260, 2014.

[17] Viveck R Cadambe, Zhiying Wang, and Nancy Lynch. Information-theoretic lower bounds on the storage cost of shared memory emulation. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '16, pages 305–314. ACM, 2016.

[18] Flavio P Calmon, Yury Polyanskiy, and Yihong Wu. Strong data processing inequalities in power-constrained gaussian channels. In *Information Theory (ISIT), 2015 IEEE International Symposium on*, pages 2558–2562, 2015.