Fully-Decentralized Coded Computing for Reliable Large-Scale Computing

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Department of Electrical and Computer Engineering

Haewon Jeong

B.S., Electrical Engineering, KAIST
M.S., Electrical an Computer Engineering, CMU

Carnegie Mellon University
Pittsburgh, PA

May, 2020

Abstract

In this thesis, I ask the question "how do we compute reliably using thousands of distributed, unreliable nodes?" We propose a system-level solution where we add redundant data across distributed nodes using the technique called *"coded computing."* Our main contribution is developing strategies for a masterless, fully-decentralized setting for important computation primitives in machine learning (ML) and scientific computing applications while minimizing the overhead of coding. For distributed matrix multiplication, we make a fundamental advance by proposing coded computing strategies that outperform prior works by an unbounded factor, including recently-developed coded computing strategies as well as traditional Algorithm-Based Fault Tolerance (ABFT) strategies. We also propose coded computing schemes for other primitives such as fast Fourier transform (FFT) and matrix QR factorization.

Completing computation reliably and in time under diverse unpredictabilities (e.g., stragglers, node failures, bit flips) is becoming a more important problem. The amount of data we collect is growing exponentially and recent developments in ML have enabled utilizing and processing such large quantities of data. This has not only led to an increase in the scale of computing but also the wide popularity of large-scale computing across our society. I will discuss how masterless coded computing can be a more efficient fault-tolerance technique under growing unpredictability in computing systems, providing both theoretical and experimental evidence.

Acknowledgments

I can say with a lot of gratitude that my journey of getting a Ph.D. degree was largely fun and enjoyable. I believe that this was mostly thanks to my advisor, Prof. Pulkit Grover, who is the most positive and supportive advisor one can ask for. His passion for science and research constantly motivated me and his thoughtful guidance built my confidence as a researcher. Another big part of the joy in my Ph.D. journey was the wonderful collaborators I had: Ziqian Bai, Dr. Chris Blake, Prof. Viveck Cadambe, Malhar Chaudhari, Sanghamitra Dutta, Dr. Christian Engelmann, Mohammad Fahim, Vipul Gupta, Farzin Haddapour, Dr. Jukka Kohonen, Yuqiu Liu, Prof. Tze Meng Low, Quang Minh Ngyuen, Prof. Kannan Ramchandran, Prof. Teemu Roos, Utsav Sheth, Yuk Wong, Dr. Yaoqing Yang, Dr. Fangwei Ye. Especially, I want to mention that Prof. Viveck Cadambe, Prof. Tze Meng Low, and Dr. Christian Engelmann gave me so much valuable inputs and support as if they are my co-advisors. I want to mention Prof. Virginia Smith and Prof. Alex Dimakis who graciously agreed to be on my thesis committee and carved our their time to help me shape this thesis. I am also grateful to my friends, both new friends I made in Pittsburgh and old friends from Korea, for giving me moral support whenever I needed. Finally, I want to thank my family for always supporting my goals and cheering me on.

This work was sponsored by: NSF CAREER (NSF-1350314), NSF EARS (NSF-1343324) NSF CIF-1763561, NSF WiFiUS (1702694), and SRC SONIC center.

Thesis Committee Members:

Pulkit Grover (Chair)
Alexandros G. Dimakis
Christian Engelmann
Tze Meng Low
Virginia Smith

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Reliability Issues in Computing

"Hardware Problems on October 14 and 15. The 360/91 was down from 4:50 P.M. Tuesday until 10:45 A.M. Wednesday because of a hardware failure due to a faulty SLT card in the floating point section of the CPU. The backup card had already been used. The IBM Emergency Parts Center located one Tuesday night in Palo Alto, but it was damaged in transit. Another card was located in Pennsylvania; it was due to arrive in Los Angeles by 6:30 P.M. Wednesday, but the IBM Customer Engineer succeeded in repairing the damaged card and got the 91 up by 11 A.M., averting an additional 9.5 hours of downtime."

CCN Newsletter, University of California, Los Angeles, Oct. 15, 1975

"In 2003 in Schaerbeek, Belgium, an single-event upset (SEU) was responsible for giving a candidate in an election an extra 4,096 votes. This was only spotted because it meant the politician concerned had more votes than it was possible to get and an investigation ensued."

The Independent, Feb. 17, 2017

"The Spaceborne Computer Returns to Earth. The computer encountered a variety of anomalies, ranging from temperature anomalies to higher rates of processor cache errors to an astronaut's knee bumping into the emergency power switch and causing a hard crash."

HPC Wire, June 10, 2019

It is easy to assume that computers are purely logical machines that take inputs, follow through pre-determined operations, and produce outputs, and to forget about the physical reality underneath the logical operations. Computers are still physical systems that follow the laws of physics. Albeit this sounds like a self-evident statement, it has an important implication: computers are subject to the statistical nature of particles, and hence subject to *"noise"*. One might question if computing can transcend the physical restrictions at all. However, through Szilard's machine [68, 105], it was shown that information is fundamentally a physical quantity, and hence the processing of information is a physical process.

The question of how to assemble physical components together to output reliable computation results under inherent noise in nature has been an important thread in computer science research since the beginning of the field. The pioneer of modern computer architecture, John von Neumann initiated the discussion on this topic in a series of five lectures at Caltech [115] in 1952, and it attracted tremendous attention from prominent researchers [10, 122] including the founder of Information Theory, Claude Shannon [79].

While the deep-rooted problem of reliability in computing remains a fundamental issue, how it manifests depends on the physical substrate and the specifics of a computing system. Starting from vacuum tube computers in the 1940s, computing technology has gone through a myriad of revolutionary changes, and factors that affect the reliability of computing have been evolving at a fast pace. There are more diverse factors than one might imagine that could cause faulty computing. At the atomic level, cosmic rays, the flux of high-energy particles that come from outer space, could interfere with atoms in a chip and cause a bit flip. Computing failures can also result from faulty read from aging memory. For instance, flash memories have a finite read/write endurance, which means that after some number of cycles, a flash cell becomes unreliable and

unable to retain the stored information. Unexpected power outages can also bring important computation to a sudden halt[1]. Software errors such as bugs for an unconsidered corner case comprise a large fraction of computer crashes.

An important change in computing paradigms in recent decades is scaling out instead of scaling up. As we are reaching the end of Moore's law and Dennard's scaling, going massively parallel became a more efficient solution for scaling computation. There are unreliability issues that arise from massive parallelism. Firstly, since we are concurrently using tens of thousands of processors, each of which has its own failure probability, as a whole, the probability of failure increases and mean-time-between-failure (MTBF) decreases. Consider the Fugaku supercomputer that is now being built in Japan to be available in 2021. The system will have 150,000 physical nodes with a total of 8 million cores [73]. For a system-level mean-time-between-failures (MTBF) of 24-48 hours, the MTBF of each node must be 411-822 years. Such nodes are difficult to design, implement, and test, and provide little-to-no room for unexpected reliability issues (e.g. dirty power, unexpected early wear-out [39, 44]) that have been experienced in the past. Unreliability is not limited to hard crashes or soft errors; unpredictable program execution time is also a reliability issue since users want to obtain computation results within the expected time. The issue of unpredictably slow compute nodes, known as *"stragglers"*, are well-recognized in cloud computing literature [1, 22, 49]. It is an increasing concern as we increase the number of compute nodes because each node exhibits different performance in practice (even identical ones) and it becomes harder to predict the job completion time with thousands of intrinsically heterogeneous nodes. Another factor that compounds the problem is that these large-scale systems are often multi-tenant. As there are dynamically changing job requests, it is not always possible to do optimal job distribution. Sub-optimal scheduling of jobs can lead to long job queues at certain compute nodes or network congestion, which all contribute to unpredictable program execution time.

[1] At Oak Ridge National Lab, which is on the wildlife reserve, wild animals running over power lines have led to power outages several times.

The problem of reliable computing will continue to diversify as new technology and paradigm of computing emerge. To go beyond the limit of transistor-based computing technology, people are exploring completely new technologies such as quantum computing or biological computing. In the early phase of these new technologies, one of the biggest challenges would be providing high reliability and fidelity. Even with the same hardware technology, how we utilize and service computing technologies is going through innovations. For example, federated learning, in which edge devices (e.g., smartphones) and the central server communicate back and forth to train a private machine learning model, has been an exciting field of study [72]. A critical challenge in federated learning is dealing with unreliable edge devices that can drop out of computation due to connectivity or energy constraints. Another upcoming idea is *serverless computing* [58] offered by cloud providers where a user can run applications at a lower cost by not having a dedicated server but instead using machine resources that are dynamically allocated based on each provider's policy. A user of serverless computing service cannot know which machine will become available or unavailable, and incorporating such dynamically-changing compute resources can be a new unreliability problem.

1.2 Coded Computing as a Low-Overhead Reliability Technique

Despite the effort for assuring the reliability of each component in computing systems, there remain uncontrollable factors. What are the techniques used to handle random unreliabilities in today's large-scale computing systems? All current production-quality technologies rely on *checkpointing*, where we store the snapshot of computation at a regular interval and roll back to the most recent checkpoint in case of failure. To store the synchronized state of distributed nodes, checkpoints are often stored in a shared parallel file system. The time spent in checkpointing is significant (15–30 minutes in 20009) because of the I/O burden to the parallel file system [13].

There has been an active research to reduce the overhead of checkpointing, such as reducing the size of checkpoints [9, 53] or in-memory checkpointing [133].

Another reliability technique often considered in distributed computing systems is replication [2, 5, 33, 34, 117]. In replication strategies, we create replicas of the same process so that even when a node (or a process) fails, we can proceed with one of the surviving replicas, without having to roll back and restart. Replication has large resource overhead (at least 2x) as we have to use limited computing resources to perform identical computations just for reliability. However, recent studies have shown that using replication can be more efficient than checkpoint-restart in systems with small MTBF [34], and using replication along with checkpoint-restart can greatly reduce mean-time-between-interruption (MTBI) [5].

Checkpoint-restart and replication can be applied universally, agnostic to the computation task. Can we sacrifice the universality and come up with a more application-specific reliability method to reduce the time and resource overheads of these generic strategies? Especially, can we borrow ideas from information theory, that has served as the foundation of modern digital communication by contriving a mathematical tool to design redundancy that is unboundedly more efficient than replication? This is the approach we propose in this thesis, called *"coded computing"*, where we add redundancies through encoding and decoding using the ideas from error-correcting codes. We will explain the basic idea of coded computing in more detail in Chapter 2. In this section, we provide a brief history of the concept of coded computing, which is almost as old as the problem of reliable computing.

In 1958, Elias studied if we can extend Shannon's noisy coding theorem in the landmark paper, *"Communication in the presence of noise"* [100], to noisy AND gates, and wrote a paper titled *"Computation in the presence of noise"* [30]. Subsequent works also studied how to incorporate noiseless encoders and decoders to build reliable Boolean gates [86, 121, 122]. A more advanced encoding and decoding technique called algorithm-based fault tolerance (ABFT) was proposed by Huang and Abraham in 1984 [52] to detect and correct errors on circuits during

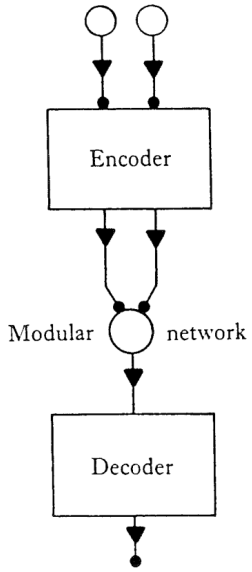


Figure 1.1: Computation system with noiseless encoders and decoders to make unreliable gates more reliable. [122, Figure 4.8]

linear algebra operations and later developed for other computations such as fast Fourier transform (FFT) [18, 92]. Chen and Dongarra discovered that the ABFT technique could be used for parallel matrix algorithms for HPC systems [17] to deal with node failures. This has initiated extensive research in ABFT [8, 21, 47, 128], and soft error detection/correction using ABFT was also studied [16, 78].

In 2015, the same idea was proposed by Lee et al. to combat the straggler problem [67], and was given the name "coded computing". Since the pioneering work by Lee et al., coded computing has generated exciting results including: coding strategies for distributed optimization [61, 91, 108, 129], addressing von Neumann's 60-year-old question of error-resilient neural network training [28], obtaining storage-optimal solutions to error-resilient matrix-multiplication [29, 101, 132], and obtaining the first solution to linear transforms with all elements being error-prone [125].

Compared to existing fault tolerance techniques, coded computing can be a much more efficient solution in terms of system overhead. Coded computing does not require roll-back or

restart to recover the lost result from a failed node. Instead, it requires communication from the surviving nodes and low-complexity decoding operation to recover the computation output. Also, compared to replication-based schemes, coded computing requires much less redundancy. Replication-based schemes require 2x redundancy for detecting an error and 3x redundancy (also known as triple modular redundancy (TMR)) for correcting an error. Coded computing, on the other hand, can provide single error detection and correction capability with a small (asymptotically negligible) redundancy.

1.3 Definitions and Notation

Computation System Models

We will mainly use two models of distributed computation system in this thesis: a master-worker setup and a masterless setup.

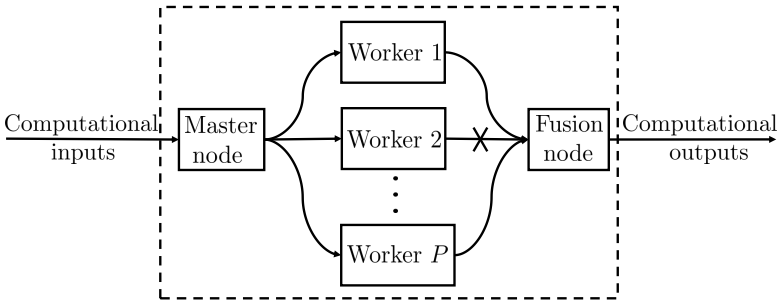


Figure 1.2: A computational system: The master node receives the computational inputs and sends appropriate tasks to the workers. The workers are prone to faults and delays. The fusion node aggregates the computational outputs from the subset of successful workers and produces the desired computational outputs.

Definition 1.3.1. [Master-Worker Setup] In the master-worker setup, we assume that there are three types of nodes: (i) a master node; (ii) worker nodes; and (iii) a fusion node whose roles are the following:

(i) A *master node* that receives computational inputs and perform pre-processing if required. It then distributes (pre-processed) input data to worker nodes.

(ii) *Worker nodes* perform the given computation on the input it received from a master node. A *successful worker* sends the resulting computation to the fusion node. A *failed* worker does not send the result to the fusion node.

(iii) A *fusion node* that receives outputs from the subset of successful worker nodes. If a fusion node receives enough number of successful workers, it will perform post-processing (e.g., decoding) and produces the final computational output. Otherwise, it declares a "computation failure."

Note that we make a distinction between a master node and a fusion node as they serve different functionalities. However, the distinction is more logical, and both master node and fusion node will reside in one physical node. Sometimes, we will omit this distinction, and call a fusion node as a master node in some places. Details of each node's role would depend on the computation goal.

Definition 1.3.2. [Masterless Setup] A masterless setup consists of a set of identical compute nodes. There is no central node present during the computation and nodes do not have any shared memory. Data located at different nodes can be shared only through explicit communication between two nodes. We assume a fully-connected network where any worker node can communicate with any other node in the system directly.

Also, we will use the term *nodes* and *processors* interchangeably in this document. In real-world distributed systems, *nodes* are composed of multiple *processors*, and a processor has multiple *cores*. While we will acknowledge these differences in Chapter 4 where we discuss experiments, in other places, one should regard *nodes* as a more abstract notion.

Error-Correcting Codes

We will very briefly review the basics of error-correcting codes as they have been designed for communication and storage systems. For more comprehensive understanding, we refer the readers to excellent textbooks including [74, 94, 95].

The objective of error-correcting codes is to add redundancy on the given data to recover lost or corrupted data through redundancy. Mathematically, this can be described as follows. Let $\mathbf{m} \in \mathbb{F}^k$ denote a length-k message vector. By adding $n-k$ redundant symbols, we want to encode this message vector into a length-n code vector $\mathbf{x} \in \mathbb{F}^n$. The encoding function:

$$\mathcal{E} : \mathbb{F}^k \to \mathbb{F}^n \tag{1.1}$$

that maps $\mathbf{m}$ to $\mathbf{x}$ can be any function, but in this work, we will only consider linear functions. This is referred to as *linear encoding*. Given that $\mathcal{E}$ is linear, now the encoding process can be represented as:

$$\mathbf{x} = \mathbf{m}G, \tag{1.2}$$

where G is a k-by-n matrix, called *a generator matrix*. One important parameter in error-correcting codes is a code rate $R = k/n$ that represents how much portion of $\mathbf{x}$ contain the original information.

A crucial question to ask is: how many lost symbols can we recover if we add $n-k$ redundant symbols? A reasonable hope would be tolerating $n-k$ erasures since we added $n-k$ more symbols. This is indeed the provably best performance any encoding function can achieve, and there exists a linear encoding scheme that achieves this. Codes that achieve this are called maximum distance separable (MDS) codes.

Systematic codes under the linear encoding (1.2) are codes that have a generator matrix of the form:

$$G = \left[\begin{array}{c|c} I_{k\times k} & P \end{array}\right]. \tag{1.3}$$

The left k-by-k square block is identity matrix, followed by a k-by-$(n-k)$ parity generating matrix P. This means that the first k symbols of $\mathbf{x}$ would be just a copy of the original message $\mathbf{m}$, and the last $n-k$ symbols would be linear combinations of $\mathbf{m}$ encoded by: $\mathbf{m}P$. In systematic codes, we will call the encoded symbols *parity symbols* or *checksums*.

Now, we introduce some notions that are closely related to distributed computing settings.The *recovery threshold* is the minimum number of successful workers required by the fusion node to recover the computation output. We will denote the recovery threshold by K. The *recovery bandwidth* is the minimum number of symbols to be communicated to the fusion node to recover the computation output.

Failure Models

Largely, the failure model we consider in this thesis is *"an erasure model"* where we assume that when a node fails, we lose the entire data it held. An erasure can happen for various reasons. It could be due to *fail-stop errors* which is a commonly used abstraction for failures in HPC to describe a situation where process behavior becomes arbitrary. It could be also due to *stragglers*. If a process does not respond within the set deadline, it could be considered as an erasure. Also, we assume that a node failure can be isolated[2].

Latency Models

We use the α-β model to estimate the point-to-point communication cost. In the α-β model, the time to send or receive a message of s bytes is :

$$T = \alpha + s \cdot \beta \tag{1.4}$$

Here, α is startup time to establish a connection between two nodes, and β is the bandwidth cost required to transfer one symbol. For an algorithm that requires multiple rounds of message

[2]One failure can easily trickle down to other nodes and isolating a failure is not always straightforward in real-world systems. However, we limit ourselves to a simple model in this thesis.

exchanges, total communication time can be written as follows:

$$T = C_1\alpha + C_2\beta, \tag{1.5}$$

where C_1 is the number of communication rounds, C_2 is the number of symbols communicated in a sequence. To be more precise, if we denote b_i as the maximum number of symbols communicated between two nodes at the i-th round, C_2 can be written as:

$$C_2 = \sum_{i=1}^{C_1} b_i. \tag{1.6}$$

This is because the next round does not start until the previous round is completed, and the bandwidth latency for each round is dominated by the largest message. Symbols can have different units, such as bits or bytes, but in this work we do not specify any units.

In some places, we will use the α-β-γ model to incorporate the computation cost into equation:

$$T = \alpha C_1 + \beta C_2 + \gamma C_3, \tag{1.7}$$

where C_3 is the number of floating point operations (flops).

1.4 Main Contributions and Outline

This thesis considers a long-standing intellectual problem of computing reliably with unreliable components adapted to the present-day computing systems by marrying large-scale distributed algorithms and coding theory. Main contributions of this thesis are as follows:

- We propose MatDot codes for matrix multiplication that advance on the existing literature in ABFT and coded computing strategies in terms of *recovery threshold*. It was later proven that the recovery threshold of MatDot codes is optimal under input storage constraint [132]. We also constructed PolyDot codes that can flexibly trade off communication cost and storage cost. (Chapter 2)

- We argue that coded computing with a master node is not scalable (with some preliminary experimental evidence), and we introduce the idea of *"masterless coded computing"*, where data pre- and post-processing are also done without a central master node. (Chapter 3)
- An important topic to be thought over in masterless coded computing is reducing the communication cost of distributed encoding and decoding. By borrowing the idea of locally-recoverable (LRC) codes from the latest distributed storage codes literature, we propose LRC coded matrix multiplication which allows for more communication-efficient recovery in case of a single failure. (Chapter 3.1)
- We propose fully-distributed coded computing algorithms for existing numerical algorithms that are extensively used in a broad set of HPC applications such as: Scalable Universal Matrix Multiplication Algorithm (SUMMA) and the 4-step algorithm for FFT. (Chapter 3.2)
- Finally, we show experimental evaluation of the proposed 3D Coded SUMMA on a HPC system. Through extensive experiments, we compare when coded computing can outperform existing fault-tolerance techniques such as replication or ABFT. (Chapter 4)

1.4.1 Excluded Work

During my Ph.D., I also worked on *energy-adaptive error-correcting codes* which is not included in this thesis. The goal of this research was to design an error-correcting code that can adapt to time-varying environments (e.g., SNR, energy constraints) to minimize energy consumption for encoding/decoding. I proposed two novel designs of energy-adaptive codes: (1) energy-adaptive polar codes (theoretical analysis), (2) energy-adaptive LDPC codes (simulation analysis). For interested readers, we refer to [54, 55, 57].

Chapter 2

Introduction to Coded Computing

"It seems somewhat strange to be writing a paper on parallel matrix multiplication almost two decades after commercial parallel systems first became available. One would think that by now we would be able to manage such an apparently straightforward task with simple, highly efficient implementations. Nonetheless, we appear to have gained new insight into this problem."

The first paragraph of "SUMMA: scalable universal matrix multiplication algorithm" by R. A. van de Geijn and J. Watts, 1997

This chapter will be a gentle introduction to coded computing. We will first walk through the basic concept of coded computing by showing how we can apply coding to distributed matrix multiplication since matrix multiplication is not only a crucial building block of numerical algorithms but also straightforward to understand. After providing a few simple examples in Chapter 2.1.1, we illustrate MatDot codes and PolyDot codes more formally, which are our major breakthroughs in coded matrix multiplication. Then, in Chapter 2.2, we take one step back from coded matrix multiplication, and discuss a more global view on how coded computing can be applied to other classes of computation.

2.1 Coded Matrix Multiplication

Throughout this chapter, we will assume the master-worker setup defined in 1.3 and our computation goal is to compute the following matrix multiplication:

$$\mathbf{C} = \mathbf{A}\mathbf{B}, \tag{2.1}$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are assumed to be N-by-N square matrices for simplicity[1]. The key idea of coded matrix multiplication is encoding redundancy on the inputs $\mathbf{A}$ and $\mathbf{B}$ before the computation so that the computation output $\mathbf{C}$ can be protected from any possible failures during computation. We believe that this would be best understood through simple examples.

Below are a few more notations we use throughout this chapter:

- P: The total number of worker nodes used.
- m: The storage parameter that denotes that a fixed $1/m$ fraction of each of the input matrices can be stored at each node.
- k: The recovery threshold of a coding strategy.

2.1.1 Simple examples

We provide simple examples of three different coded matrix multiplication strategies: (i) ABFT matrix multiplication [52] (also called *Product codes* in [67]), (ii) Polynomial codes [130] and then (iii) our proposed construction, MatDot codes. We will evaluate the straggler tolerance of a strategy by its recovery threshold, k. For all the examples, we consider the simplest case with $m = 2$. Let us begin by describing the first strategy, ABFT matrix multiplication.

Example 2.1.1 (ABFT codes [52] ($m = 2$, $k = 2\sqrt{P}$))**.** *Consider two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ that are split as follows:*

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_0 & \mathbf{B}_1 \end{bmatrix}$$

[1]Coding strategies we introduce here can be extended to rectangular matrices as well.

where $\mathbf{A}_0, \mathbf{A}_1$ *are sub-matrices (row-blocks) of* $\mathbf{A}$ *of dimension* $N/2 \times N$ *and* $\mathbf{B}_0, \mathbf{B}_1$ *are sub-matrices (column-blocks) of* $\mathbf{B}$ *of dimension* $N \times N/2$*. Using ABFT, it is possible to compute* $\mathbf{AB}$ *over* P *nodes such that,* $(\mathbf{i})$ *each node uses* $N^2/2$ *linear combinations of the entries of* $\mathbf{A}$ *and* $N^2/2$ *linear combinations of the entries of* $\mathbf{B}$ *and* $(\mathbf{ii})$ *the overall computation is tolerant to* $P - 2\sqrt{P}$ *stragglers in the worst case. Thus, any* $P - (P - 2\sqrt{P}) = 2\sqrt{P}$ *worker nodes suffice to recover* $\mathbf{AB}$*.*

ABFT codes use the following strategy: P *processors are arranged in a* $\sqrt{P} \times \sqrt{P}$ *grid. ABFT codes encode two row-blocks of* $\mathbf{A}$ *and two column-blocks of* $\mathbf{B}$ *separately using two systematic* $(\sqrt{P}, 2)$ *MDS codes. Then, we distribute the* i*-th encoded row-block of* $\mathbf{A}$ *to all the worker nodes on the* i*-th row of the grid, and the* j*-th encoded column-block of* $\mathbf{B}$ *to all the worker nodes on the* j*-th column of the grid. Note that here the grid indexing is* $i = 1, 2, \ldots, \sqrt{P}$ *and* $j = 1, 2, \ldots, \sqrt{P}$*. An example for* $P = 9$ *is shown in Fig. 2.1. The worst case arises when all but one worker node in the lower right* $(\sqrt{P} - 1) \times (\sqrt{P} - 1)$ *part of the grid fail. Thus, the worst case recovery threshold is* $P - (\sqrt{P} - 1)^2 + 1 = 2\sqrt{P}$*. For the example given in Fig. 2.1 where* $P = 9$*, recovery threshold is* $2\sqrt{P} = 6$*.* ■

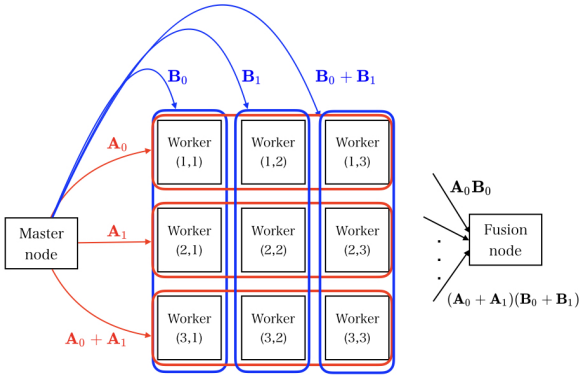


Figure 2.1: ABFT matrix multiplication [52] for $P = 9$ worker nodes with $m = 2$, where the recovery threshold is 6.

In the previous example, the recovery threshold was a function of P and thus it requires more successful worker nodes as we use more processors. However, as we will show in the next

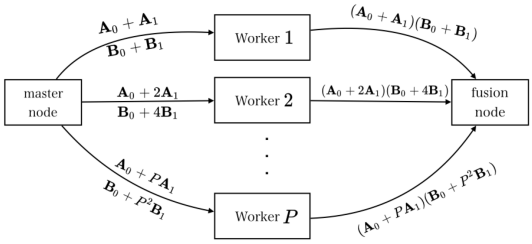


Figure 2.2: Polynomial Codes [130] with $m = 2$. The recovery threshold is 4.

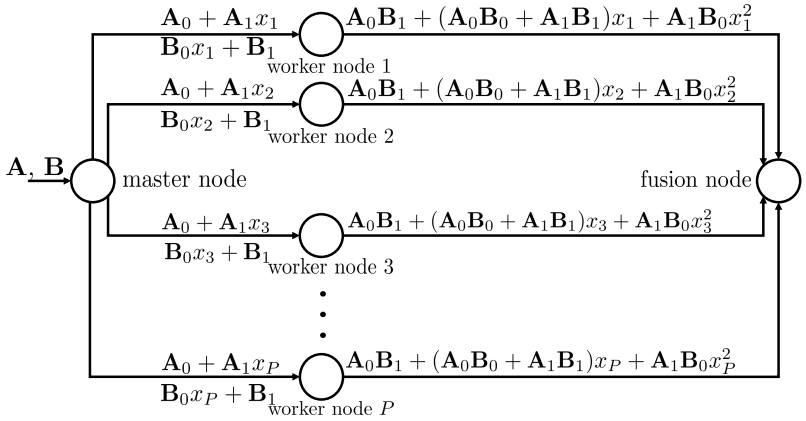


Figure 2.3: An illustration of the computational system with four worker nodes and applying MatDot codes with $m = 2$. The recovery threshold is 3.

example, Polynomial codes [130] provide a superior recovery threshold that does not depend on P.

Remark 2.1.1. In the worst-case ABFT codes might require $\Theta(\sqrt{P})$ nodes to finish, but in the best-case only m^2 nodes might suffice, e.g., if all the systematic nodes finish first. Therefore, some specific subsets of nodes of size smaller than the recovery threshold can sometimes suffice for reconstruction, even though not all subsets of this size suffice. For a detailed discussion on best-case and average-case recovery, the reader is referred to [67].

Example 2.1.2 (Polynomial codes [130] ($m = 2, k = 4$))**.** *Consider two* $N \times N$ *matrices* $\mathbf{A}$ *and*

$\mathbf{B}$ *that are split as follows:*

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_0 & \mathbf{B}_1 \end{bmatrix}.$$

Polynomial codes compute $\mathbf{AB}$ *over* P *nodes such that,* $(\mathbf{i})$ *each node uses* $N^2/2$ *linear combinations of the entries of* $\mathbf{A}$ *and* $N^2/2$ *linear combinations of the entries of* $\mathbf{B}$ *and* $(\mathbf{ii})$ *the overall computation is tolerant to* $P-4$ *stragglers, i.e., any* 4 *nodes suffice to recover* $\mathbf{AB}$*. Polynomial codes use the following strategy: Node* i *computes* $(\mathbf{A}_0+\mathbf{A}_1 i)(\mathbf{B}_0+\mathbf{B}_1 i^2), i = 1, 2, \ldots P$*, so that from any* 4 *of the* P *nodes, the polynomial* $p(x) = (\mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_0 x + \mathbf{A}_0\mathbf{B}_1 x^2 + \mathbf{A}_0\mathbf{B}_1 x^3)$ *can be interpolated. Having interpolated the polynomial,* $\mathbf{AB}$ *as* $\begin{bmatrix} \mathbf{A}_0\mathbf{B}_0 & \mathbf{A}_0\mathbf{B}_1 \\ \mathbf{A}_1\mathbf{B}_0 & \mathbf{A}_1\mathbf{B}_1 \end{bmatrix}$ *can be obtained from the coefficients (matrices) of the polynomial.* ■

Finally, we show an example of our novel *MatDot* code construction that achieves a smaller recovery threshold as compared with Polynomial codes. Unlike ABFT and Polynomial codes, MatDot divides matrix $\mathbf{A}$ vertically into column-blocks and matrix $\mathbf{B}$ horizontally into row-blocks.

Example 2.1.3. *[MatDot codes (*$m = 2$*,* $k = 3$*)]*

MatDot codes compute $\mathbf{AB}$ *over* P *nodes such that,* $(\mathbf{i})$ *each node uses* $N^2/2$ *linear combinations of the entries of* $\mathbf{A}$ *and* $N^2/2$ *linear combinations of the entries of* $\mathbf{B}$ *and* $(\mathbf{ii})$ *the overall computation is tolerant to* $P-3$ *stragglers, i.e.,* 3 *nodes suffice to recover* $\mathbf{AB}$*. The proposed MatDot codes use the following strategy: Matrix* $\mathbf{A}$ *is split vertically and* $\mathbf{B}$ *is split horizontally as follows:*

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_0 \\ \mathbf{B}_1 \end{bmatrix}, \tag{2.2}$$

where $\mathbf{A}_0, \mathbf{A}_1$ *are column-blocks of* $\mathbf{A}$ *of dimension* $N \times N/2$ *and* $\mathbf{B}_0, \mathbf{B}_1$ *are row-blocks of* $\mathbf{B}$ *of dimension* $N/2 \times N$*.*

Let $p_{\mathbf{A}}(x) = \mathbf{A}_0 + \mathbf{A}_1 x$ *and* $p_{\mathbf{B}}(x) = \mathbf{B}_0 x + \mathbf{B}_1$*. Let* $x_1, x_2, \cdots, x_P$ *be distinct elements in* $\mathbb{F}$*. The master node sends* $p_{\mathbf{A}}(x_r)$ *and* $p_{\mathbf{B}}(x_r)$ *to the* r*-th worker node where the* r*-th worker node*

performs the multiplication $p_{\mathbf{A}}(x_r)p_{\mathbf{B}}(x_r)$ *and sends the output to the fusion node. The exact computations at each worker node are depicted in Fig. 2.3. We can observe that the fusion node can obtain the product* $\mathbf{AB}$ *using the output of any three successful workers as follows: Let the worker nodes* $1, 2,$ *and* 3 *be the first three successful worker nodes, then the fusion node obtains the following three matrices:*

$$p_{\mathbf{A}}(x_1)p_{\mathbf{B}}(x_1) = \mathbf{A}_0\mathbf{B}_1 + (\mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_1)x_1 + \mathbf{A}_1\mathbf{B}_0x_1^2,$$
$$p_{\mathbf{A}}(x_2)p_{\mathbf{B}}(x_2) = \mathbf{A}_0\mathbf{B}_1 + (\mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_1)x_2 + \mathbf{A}_1\mathbf{B}_0x_2^2,$$
$$p_{\mathbf{A}}(x_2)p_{\mathbf{B}}(x_3) = \mathbf{A}_0\mathbf{B}_1 + (\mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_1)x_3 + \mathbf{A}_1\mathbf{B}_0x_3^2.$$

Since these three matrices can be seen as three evaluations of the matrix polynomial $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *of degree* 2 *at three distinct evaluation points* x_1, x_2, x_3*, the fusion node can obtain the coefficients of* x *in* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *using polynomial interpolation. This includes the coefficient of* x*, which is* $\mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_1 = \mathbf{AB}$*. Therefore, the fusion node can recover the matrix product* $\mathbf{AB}$*.*

■

2.1.2 MatDot and PolyDot codes

We will now provide the formal description of MatDot and PolyDot codes. We start by defining a rigorous system model.

2.1.2.1 System Model

We consider a master-worker setup given in 1.3.1, and define an (N, k, P, m) Computational system for Matrix Multiplication based on it.

Definition 2.1.1. [An (N, k, P, m) Computational system for Matrix Multiplication]

(i) A *master node* receives computational inputs, i.e., two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ and obtains, via *linear* pre-processing, $2P$ matrices as follows:

$$\widetilde{\mathbf{A}}_i = f_i(\mathbf{A}) \text{ and } \widetilde{\mathbf{B}}_i = g_i(\mathbf{B}) \text{ for } i = 1, 2, \ldots, P.$$

Here, f_i and g_i are two functions such that $f_i : \mathbb{F}^{N\times N} \to \mathbb{F}^{N/t\times N/s}$ and $g_i : \mathbb{F}^{N\times N} \to \mathbb{F}^{N/s\times N/t}$. Each $\widetilde{\mathbf{A}}_i$ for $i = 1, 2, \ldots, P$ is an $N/t \times N/s$ matrix and each $\widetilde{\mathbf{B}}_i$ for $i = 1, 2, \ldots, P$ is an $N/s \times N/t$ matrix, where s and t are two integers that satisfy $st = m$ and m is an integer that divides N. Specifically, each entry of $\widetilde{\mathbf{A}}_i$ (respectively $\widetilde{\mathbf{B}}_i$) is restricted to be an $\mathbb{F}$-linear combination[2] of the entries of $\mathbf{A}$ (respectively $\mathbf{B}$).

(ii) P *worker nodes* that perform the following operations: For $i = 1, \cdots, P$, the i-th worker node receives $\widetilde{\mathbf{A}}_i, \widetilde{\mathbf{B}}_i$ from the master node, and performs some computation on these matrices.

(iii) A *fusion node* that receives outputs from the subset of successful worker nodes.

(iv) The recovery threshold is k, i.e., a fusion node will perform post-processing if the number of successful worker is at at least k, and produces the final output $\mathbf{AB}$.

We make some informal remarks on the system model before describing our problem statement.

- For a given computation system, the parameter k is referred to as its *recovery threshold*. Note that as per the definition, the recovery threshold is a worst-case evaluation, i.e., over the worst possible choice of inputs $\mathbf{A}, \mathbf{B}$ as well as the worst set of worker failures.
- The parameter m controls the memory of each worker in the model, i.e., each worker node can store only upto a $1/m$ fraction of each of the input matrices.
- For convenience, we simply refer to an (N, k, P, m) computation system for matrix multiplication as a *computation system* in this paper; the parameters N, k, P, m can be inferred from context.
- A worker node can fail due to various reasons such as: (i) straggling due to other jobs in the queue; (ii) straggling due to network congestion; (iii) temporary unavailability (e.g.,

[2]We restrict pre-processing to be linear to capture memory constraints of each worker node. Note that, allowing for non-linear pre-processing with infinite precision can allow the master node to encode the entire input $\mathbf{A}, \mathbf{B}$ into smaller dimensional matrices over real or complex fields.

system updates or power outage). In particular, while our model states that the failed worker nodes do not send their computational outputs to the fusion node, in practice, a straggling worker node that sends its result later than an acceptable deadline may also be considered as a failure in our model. We use the term failed nodes interchangeably with the term *straggling nodes* in this paper. The parameter $P-k$ represents fault-tolerance, or equivalently, the straggler-tolerance of the system.

- Elementary coding theory also implies that an (N, k, P, m) computation systems can correct $\lfloor \frac{P-k}{2} \rfloor$ erroneous worker nodes, i.e., nodes that can output incorrect computations, though we do not focus explicitly on error correction in this paper.
- For a given computation system, the computational complexities of the master, workers, and the fusion node are referred to as the pre-processing, online, and decoding complexities. In addition to recovery thresholds, we also evaluate various computation schemes in terms of these computation complexities, as well as the communication cost from the worker nodes to the fusion node. The communication cost between the master node and worker nodes is constant in all the strategies because of the storage constraint, i.e., the master sends upto N^2/m symbols to each worker node.
- Our strategies also extend when the matrices $\widetilde{\mathbf{A}}_i$ and $\widetilde{\mathbf{B}}_i$ are allowed to be of dimensions $N/t_1 \times N/s$ and $N/s \times N/t_2$ (discussed in Remark 2.1.3 later), i.e., asymmetric storage constraints for the two inputs. Our system model also assumes that $\mathbf{A}, \mathbf{B}$ are square matrices with equal dimensions for simplicity of notation. Our ideas and results will naturally apply for cases where $\mathbf{A}, \mathbf{B}$ are non-square matrices as well, as long as the product $\mathbf{AB}$ is defined.

2.1.2.2 Problem Statement

We consider an (N, k, P, m) computation system where the computational complexities of the master, worker and fusion nodes, when evaluated in terms of parameter N, P, m, are all less than

the complexity of any sequential algorithm that takes inputs $\mathbf{A}, \mathbf{B}$ and computes the product $\mathbf{AB}$ as the output[3]. Given parameters N, P, m, among these considered systems, our problem is to determine the computation system with the smallest achievable recovery threshold.

Although the problem stated here remains open, we will present non-trivial coding strategies that achieve significantly smaller recovery threshold than previously known systems. For simplicity, we report results assuming naive matrix multiplication with complexity $\Theta(N^3)$ in our paper; our ideas and results extend, with minor modifications, to include lower complexity algorithms such as Strassen's algorithm [104].

2.1.2.3 Some Notations and Definitions

For $f(n)$ and $g(n)$ that are two functions of the variable n, $f(n) = \mathcal{O}(g(n))$ if there exists an n_0 and a constant c such that for all $n > n_0$, $f(n) \leqslant cg(n)$. Similarly, $f(n) = o(g(n))$ if for any chosen $\epsilon > 0$, one can find an n_0 such that for all $n > n_0$, $f(n) \leqslant \epsilon g(n)$. Lastly, $f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(f(n))$.

We will be using the term "row-block" to denote the sub-matrices formed when we split a matrix $\mathbf{A}$ horizontally as follows: $\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \end{bmatrix}$. Similarly, we will be using the term "column-block" to denote the sub-matrices formed when we split a matrix vertically into sub-matrices as follows: $\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_1 \end{bmatrix}$.

2.1.2.4 MatDot Code Construction

In this section, we will describe the distributed matrix-matrix multiplication strategy using MatDot codes, and then examine the computation and communication costs of the proposed strategy.

[3]The computational complexity requirement is necessary. Without this requirement, it is easy to design a $(N, k = m, P, m)$ computation system by simply storing $\mathbf{A}, \mathbf{B}$ using a (P, m) Maximum Distance Separable code at the workers, which sends the stored symbols to the fusion node which then decodes $\mathbf{A}, \mathbf{B}$ and then performs the multiplication. However, in practice, this is not parallelizing the matrix-multiplication task.

From the examples in Section 2.1.1, we have seen that for $m = 2$, the recovery threshold of MatDot codes is $k = 3$, which is lower than Polynomial codes and ABFT matrix multiplication. The following theorem shows that for any integer m, the recovery threshold of MatDot codes is $k = 2m - 1$.

Theorem 2.1.1. *For the matrix multiplication problem specified in Section 2.1.2.2 computed on the system defined in Definition 2.1.1, a recovery threshold of $2m-1$ is achievable where $m \geqslant 2$ is a positive integer that divides N.*

Before we prove Theorem 2.1.1, we first describe the construction of MatDot codes.

Construction 2.1.1. *[MatDot Codes]*

***Splitting of input matrices**: The matrix $\mathbf{A}$ is split vertically into m equal column-blocks (of N^2/m symbols each) and $\mathbf{B}$ is split horizontally into m equal row blocks (of N^2/m symbols each) as follows:*

$$\mathbf{A} = \left[\mathbf{A}_0 \ \mathbf{A}_1 \ \dots \ \mathbf{A}_{m-1}\right], \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_0 \\ \mathbf{B}_1 \\ \vdots \\ \mathbf{B}_{m-1} \end{bmatrix}, \tag{2.3}$$

where, for $i \in \{0, \dots, m-1\}$, and $\mathbf{A}_i, \mathbf{B}_i$ are $N \times N/m$ and $N/m \times N$ dimensional sub-matrices, respectively.

***Master node (encoding)**: Let $x_1, x_2, \dots, x_P$ be distinct elements in $\mathbb{F}$. Let $p_{\mathbf{A}}(x) = \sum_{i=0}^{m-1} \mathbf{A}_i x^i$ and $p_{\mathbf{B}}(x) = \sum_{j=0}^{m-1} \mathbf{B}_j x^{m-1-j}$. The master node sends to the r-th worker the evaluations of $p_{\mathbf{A}}(x), p_{\mathbf{B}}(x)$ at $x = x_r$, that is, it sends $p_{\mathbf{A}}(x_r), p_{\mathbf{B}}(x_r)$ to the r-th worker.*

***Worker nodes**: For $r \in \{1, 2, \dots, P\}$, the r-th worker node computes the matrix product $p_{\mathbf{C}}(x_r) = p_{\mathbf{A}}(x_r) p_{\mathbf{B}}(x_r)$ and sends it to the fusion node on successful completion.*

***Fusion node (decoding)**: The fusion node uses outputs of any $2m-1$ successful workers to compute the coefficient of x^{m-1} in the product $p_{\mathbf{C}}(x) = p_{\mathbf{A}}(x) p_{\mathbf{B}}(x)$ (the feasibility of this step will be shown later in the proof of Theorem 2.1.1). If the number of successful workers is smaller than $2m-1$, the fusion node declares a failure.*

Notice that in MatDot codes, we have

$$\mathbf{AB} = \sum_{i=0}^{m-1} \mathbf{A}_i \mathbf{B}_i, \tag{2.4}$$

where $\mathbf{A}_i$ and $\mathbf{B}_i$ are as defined in (2.3). The simple observation of (2.4) leads to a different way of computing the matrix product as compared with Polynomial-codes-based computation. In particular, to compute the product, we only require, for each i, the product of $\mathbf{A}_i$ and $\mathbf{B}_i$. We do not require products of the form $\mathbf{A}_i\mathbf{B}_j$ for $i \neq j$ unlike Polynomial codes, where, after splitting the matrices $\mathbf{A}, \mathbf{B}$ in to m parts, *all* m^2 cross-products are required to evaluate the overall matrix product. This leads to a significantly smaller recovery threshold for our construction.

Proof of Theorem 2.1.1. To prove the theorem, it suffices to show that in the MatDot code construction described above, the fusion node is able to reconstruct $\mathbf{C}$ from any $2m-1$ worker nodes. Observe that the coefficient of x^{m-1} in:

$$p_{\mathbf{C}}(x) = p_{\mathbf{A}}(x)p_{\mathbf{B}}(x) = \left(\sum_{i=0}^{m-1} \mathbf{A}_i x^i\right)\left(\sum_{j=0}^{m-1} \mathbf{B}_j x^{m-1-j}\right) \tag{2.5}$$

is $\mathbf{AB} = \sum_{i=0}^{m-1} \mathbf{A}_i\mathbf{B}_i$ (from (2.4)), which is the desired matrix-matrix product. Thus it is sufficient to compute this coefficient at the fusion node as the computation output for successful computation. Now, because the polynomial $p_{\mathbf{C}}(x)$ has degree $2m-2$, evaluation of the polynomial at any $2m-1$ distinct points is sufficient to compute all of the coefficients of powers of x in $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ using polynomial interpolation. This includes $\mathbf{AB} = \sum_{i=0}^{m-1} \mathbf{A}_i\mathbf{B}_i$, the coefficient of x^{m-1}. ■

In Section 2.1.2.5, we provide a complexity analysis that shows that using this strategy, the master and fusion nodes have a lower computational complexity as compared to the worker nodes in the regime where $m, P \ll N$.

2.1.2.5 Complexity Analysis of MatDot codes

Encoding/decoding complexity: Encoding for each worker requires evaluating two polynomials $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$, each of degree $m-1$, at a unique value of x where the coefficients of

these polynomials are sub-matrices of size N^2/m. We examine the encoding complexity using two algorithms here. One encoding algorithm could be to take a linear combination of m sub-matrices of size N^2/m, leading to an overall encoding complexity of $\mathcal{O}(mN^2/m) = \mathcal{O}(N^2)$ for *each worker*. Thus, the overall computational complexity of encoding for P workers is $\mathcal{O}(N^2P)$. Alternatively, one could also use fast polynomial evaluation algorithms [63, 65] which allow one to evaluate a polynomial (of degree $m-1$) at $P(>m)$ arbitrary points within a time complexity of $\mathcal{O}(P\log^2 m)$ (or more practically $\mathcal{O}(P\log^2 m \log\log m)$). Because this evaluation has to be repeated N^2/m times, the overall encoding complexity using fast polynomial evaluation algorithms becomes $\mathcal{O}\left(N^2P\frac{\log^2 m\log\log m}{m}\right)$.

Next, we examine the decoding complexity. Decoding requires interpolating the coefficient of x^{m-1} (of size N^2) in the polynomial $p_{\mathbf{C}}(x)$ of degree $2m-2$. Because we are interested in only one coefficient of the polynomial $p_{\mathbf{C}}(x)$ and not all of them, we consider the problem of inverting the corresponding Vandermonde matrix for polynomial interpolation and then computing the corresponding coefficient of x^{m-1} separately.

Let $p_{\mathbf{C}}(x) = \mathbf{C}_0 + \mathbf{C}_1 x + \ldots + \mathbf{C}_{k-1}x^{k-1}$ where $k = 2m-1$ and we are interested in interpolating only $\mathbf{C}_{m-1}$. Also, let $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k$ denote the $k(=2m-1)$ unique values at which the k fastest workers evaluated the polynomial $p_{\mathbf{C}}(x)$ and $\mathbf{V}$ denote the $k \times k$ Vandermonde matrix given by:

$$\mathbf{V} = \begin{bmatrix} 1 & \tilde{x}_1 & \tilde{x}_1^2 & \ldots & \tilde{x}_1^{k-1} \\ 1 & \tilde{x}_2 & \tilde{x}_2^2 & \ldots & \tilde{x}_2^{k-1} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & \tilde{x}_k & \tilde{x}_k^2 & \ldots & \tilde{x}_k^{k-1} \end{bmatrix}. \tag{2.6}$$

Observe that

$$(\mathbf{V} \otimes I_{N\times N}) \begin{bmatrix} \mathbf{C}_0 \\ \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{k-1} \end{bmatrix} = \begin{bmatrix} p_{\mathbf{C}}(\tilde{x}_1) \\ p_{\mathbf{C}}(\tilde{x}_2) \\ \vdots \\ p_{\mathbf{C}}(\tilde{x}_k) \end{bmatrix}$$

$$\implies \begin{bmatrix} \mathbf{C}_0 \\ \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{k-1} \end{bmatrix} = \left(\mathbf{V}^{-1} \otimes I_{N\times N}\right) \begin{bmatrix} p_{\mathbf{C}}(\tilde{x}_1) \\ p_{\mathbf{C}}(\tilde{x}_2) \\ \vdots \\ p_{\mathbf{C}}(\tilde{x}_k) \end{bmatrix}, \tag{2.7}$$

where $\otimes$ denotes the Kronecker product and $I_{N\times N}$ denotes an identity matrix of dimensions $N \times N$. The decoder first inverts the matrix $\mathbf{V}$ (complexity is at most $\mathcal{O}(k^3)$ using naive inversion algorithms[4]) and then picks the m-th row of $\mathbf{V}^{-1}$ which corresponds to the linear combination of evaluations leading to the coefficient of x^{m-1}. Next, it linearly combines these k evaluations $p_{\mathbf{C}}(\tilde{x}_1), p_{\mathbf{C}}(\tilde{x}_2), \ldots, p_{\mathbf{C}}(\tilde{x}_k)$ (of size N^2 each) using the k values in [m-th row of $\mathbf{V}^{-1}$], effectively performing the computation

$$\mathbf{C}_{m-1} = \left([m\text{-th row of } \mathbf{V}^{-1}] \otimes I_{N\times N}\right) \begin{bmatrix} p_{\mathbf{C}}(\tilde{x}_1) \\ p_{\mathbf{C}}(\tilde{x}_2) \\ \vdots \\ p_{\mathbf{C}}(\tilde{x}_k) \end{bmatrix}.$$

This second step is of complexity $\mathcal{O}(N^2k)$. Thus, the total decoding complexity is $\mathcal{O}(N^2k + k^3)$, of which, the first term dominates as we are interested in regimes where $k(= 2m-1) \ll N$.

Each worker's computational cost: Each worker multiplies two matrices of dimensions $N \times N/m$ and $N/m \times N$, requiring N^3/m operations (using standard matrix multiplication al-

[4]Note that, it might be possible to reduce the term k^3 to k^2 using improved methods of inverting Vandermonde matrices [6, 40, 62, 85, 112]. However, since this is not the dominant term in this decoding complexity analysis, we stick with the most conservative estimate k^3.

gorithms[5]). Hence, the computational complexity for each worker is $\mathcal{O}(N^3/m)$. Thus, as long as P and m are sufficiently small compared to N, *the encoding and decoding complexity is smaller than per-worker computational complexity in a scaling sense.* More specifically, for the decoding complexity to be negligible, we need $m^2 = o(N)$ (derived from $N^2(2m-1) = o(N^3/m)$). Similarly, for the encoding complexity to be negligible, we need $mP = o(N)$ (derived from $N^2P = o(N^3/m)$), again sticking to the conservative estimate of encoding complexity.

Communication cost: The master node communicates $\mathcal{O}(PN^2/m)$ symbols, and the fusion node receives $\mathcal{O}(mN^2)$ symbols from the successful worker nodes. While the master node communication cost is identical to that in Polynomial codes, the fusion node there only receives $\mathcal{O}(m^2N^2/m^2) = \mathcal{O}(N^2)$ symbols.

Remark 2.1.2. We note that in addition to communication costs, the computational cost per node is also higher for MatDot codes ($\mathcal{O}(N^3/m)$) as compared to Polynomial codes ($\mathcal{O}(N^3/m^2)$). This is suggestive of a trade-off. Thus, we also propose PolyDot codes which provide a trade-off between MatDot codes (lowest recovery threshold, higher communication and computation cost) and Polynomial codes (higher recovery threshold, lower communication and computation cost), with these two codes being its two special cases. These trade-offs are also pictorially illustrated later in Fig. 2.4 and Fig. 2.5.

Discussion on applicability of MatDot codes:

- In our recent work [101], we demonstrate the potential advantages of MatDot codes in practice. Reference [101] presents a distributed implementation of Fast approximate k-Nearest Neighbor computation using MatDot codes. The problem reduces to the online multiplication of only a set of few selected rows of a large matrix with another matrix/vector in real-time. Encoding and storing sub-matrices in advance is allowed, but the index set of rows of the first matrix is only available in the online phase. It is difficult to apply

[5]More sophisticated algorithms [104] also require super-quadratic complexity in N, and so a similar conclusion can be derived here if those algorithms are used at workers as well, as long as the complexity is super-quadratic in N.

horizontal splitting in this case as the index set of rows is not known a priori, and vertical splitting of the first matrix, as done in MatDot codes, is better suited.

- In several large-scale computing settings, storage is the primary cause that necessitates parallelizing or distributing the computation across multiple nodes. The actual computation cost is often cheap, and in fact often cheaper than communication costs too. The main cause of latency or straggling is attributed to several factors, which also include queuing of other tasks or limitations of communication bandwidth [116, 117]. Thus, the actual time that each worker node takes is a combination of three terms: the delay-free computation cost, the delay-free communication cost and the unpredictable delay or straggling, which could even be higher than the first two terms depending on the nature of the queuing in the system. In several models in existing literature, the total time has also been modeled with distributions which do not depend on the computation cost or communication cost [116, 117]. In such scenarios, MatDot codes would be significantly beneficial in reducing latency as compared to existing techniques as it requires the fusion node to wait for the fewest workers. Alternatively, when the computation and communication costs dominate storage costs, one could use Polynomial codes, or interpolate between these two codes using our proposed PolyDot framework (see Section 2.1.2.6).
- MatDot codes can also be written in a systematic form. See Chapter 3.1.2.

2.1.2.6 PolyDot Code Constructions

In this section, we present a code construction, named *PolyDot codes*, that provides a trade-off between per-worker computation/communication costs and recovery thresholds. Polynomial codes [130] have a higher recovery threshold of m^2, but have a lower per-worker computation cost of $\mathcal{O}(N^3/m^2)$ and communication cost of $\mathcal{O}(N^2/m^2)$ per worker node. On the other hand, MatDot codes have a lower recovery threshold of $2m-1$, but have a higher per-worker computation cost of $\mathcal{O}(N^3/m)$ and a higher communication cost of $\mathcal{O}(N^2)$ per-worker. This section

constructs a code that bridges the gap between Polynomial codes and MatDot codes so that we can get intermediate per-worker computation/communication costs and recovery thresholds, with Polynomial and MatDot codes being two special cases. To achieve this goal, we propose PolyDot codes, which may be viewed as an interpolation of MatDot codes and Polynomial codes, with one extreme being MatDot codes and the other extreme being Polynomial codes.

We follow the same problem setup and system assumptions as in MatDot codes. In the following theorem, we obtain the recovery threshold achieved by PolyDot codes.

Theorem 2.1.2. *For the matrix multiplication problem specified in Section 2.1.2.2 computed on the system defined in Definition 2.1.1, there exist codes with a recovery threshold of $t^2(2s-1)$ and a communication cost from each worker node to the fusion node bounded by $\mathcal{O}(N^2/t^2)$ for any positive integers s, t such that $st = m$ and both s and t divide N.*

Before we move on to describe the PolyDot code construction and prove Theorem 2.1.2, we first introduce PolyDot codes with a simple example for $m = 4$ and $s = t = 2$.

Example 2.1.4. *[PolyDot codes ($m = 4, s = 2, k = 12$)]*

Matrix $\mathbf{A}$ is split into sub-matrices $\mathbf{A}_{0,0}, \mathbf{A}_{0,1}, \mathbf{A}_{1,0}, \mathbf{A}_{1,1}$, each of dimension $N/2 \times N/2$. Similarly, matrix $\mathbf{B}$ is split into sub-matrices $\mathbf{B}_{0,0}, \mathbf{B}_{0,1}, \mathbf{B}_{1,0}, \mathbf{B}_{1,1}$ each of dimension $N/2 \times N/2$ as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{0,0} & \mathbf{B}_{0,1} \\ \mathbf{B}_{1,0} & \mathbf{B}_{1,1} \end{bmatrix}. \tag{2.8}$$

Notice that, from (2.8), the product $\mathbf{AB}$ can be written as

$$\mathbf{AB} = \begin{bmatrix} \sum_{i=0}^{1} \mathbf{A}_{0,i}\mathbf{B}_{i,0} & \sum_{i=0}^{1} \mathbf{A}_{0,i}\mathbf{B}_{i,1} \\ \sum_{i=0}^{1} \mathbf{A}_{1,i}\mathbf{B}_{i,0} & \sum_{i=0}^{1} \mathbf{A}_{1,i}\mathbf{B}_{i,1} \end{bmatrix}. \tag{2.9}$$

Now, we define the encoding functions $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ as

$$p_{\mathbf{A}}(x) = \mathbf{A}_{0,0} + \mathbf{A}_{1,0}x + \mathbf{A}_{0,1}x^2 + \mathbf{A}_{1,1}x^3,$$

$$p_{\mathbf{B}}(x) = \mathbf{B}_{0,0}x^2 + \mathbf{B}_{1,0} + \mathbf{B}_{0,1}x^8 + \mathbf{B}_{1,1}x^6.$$

Observe the following:

(i) the coefficient of x^2 *in* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *is* $\sum_{i=0}^{1} \mathbf{A}_{0,i}\mathbf{B}_{i,0}$*,*

(ii) the coefficient of x^8 *in* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *is* $\sum_{i=0}^{1} \mathbf{A}_{0,i}\mathbf{B}_{i,1}$*,*

(iii) the coefficient of x^3 *in* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *is* $\sum_{i=0}^{1} \mathbf{A}_{1,i}\mathbf{B}_{i,0}$*, and*

(iv) the coefficient of x^9 *in* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *is* $\sum_{i=0}^{1} \mathbf{A}_{1,i}\mathbf{B}_{i,1}$*.*

Let $x_1, \cdots, x_P$ *be distinct elements of* $\mathbb{F}$*. The master node sends* $p_{\mathbf{A}}(x_r)$ *and* $p_{\mathbf{B}}(x_r)$ *to the* r*-th worker node for* $r \in \{1, \cdots, P\}$*. The* r*-th worker node performs the multiplication* $p_{\mathbf{A}}(x_r)p_{\mathbf{B}}(x_r)$ *and sends the result to the fusion node.*

Let worker nodes indexed from 1 *to* 12 *be the first* 12 *worker nodes that send their results to the fusion node. Then the fusion node obtains the matrices* $p_{\mathbf{A}}(x_r)p_{\mathbf{B}}(x_r)$ *for all* $r \in \{1, \cdots, 12\}$*. Since these* 12 *matrices are essentially twelve distinct evaluations of the matrix polynomial* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *of degree* 11 *at twelve distinct points* $x_1, \cdots, x_{12}$*, the coefficients of the matrix polynomial* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *can be obtained using polynomial interpolation. This includes the coefficients of* x^{i+2+6j} *for all* $i, j \in \{0, 1\}$*, i.e.,* $\sum_{k=0}^{1} \mathbf{A}_{i,k}\mathbf{B}_{k,j}$ *for all* $i, j \in \{0, 1\}$*. Once the matrices* $\sum_{k=0}^{1} \mathbf{A}_{i,k}\mathbf{B}_{k,j}$ *for all* $i, j \in \{0, 1\}$ *are obtained, the product* $\mathbf{AB}$ *is obtained by (2.9).* ■

The recovery threshold for $m = 4$ in Example 2.1.4 is $k = 12$. This is larger than the recovery threshold of MatDot codes, which is $k = 2m - 1 = 9$, and smaller then the recovery threshold of Polynomial codes, which is $k = m^2 = 16$. Hence, we can see that the recovery thresholds of PolyDot codes are between those of MatDot codes and Polynomial codes.

Construction 2.1.2 describes the general construction of PolyDot(m, s, t) codes. Note that, although two parameters m and s are sufficient to characterize a PolyDot code, we include t in the parameters for better readability.

Construction 2.1.2. *[PolyDot(m, s, t) codes]*

***Splitting of input matrices**: $\mathbf{A}$ and $\mathbf{B}$ are split both horizontally and vertically:*

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0,0} & \cdots & \mathbf{A}_{0,s-1} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{t-1,0} & \cdots & \mathbf{A}_{t-1,s-1} \end{bmatrix},$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{0,0} & \cdots & \mathbf{B}_{0,t-1} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{s-1,0} & \cdots & \mathbf{B}_{s-1,t-1} \end{bmatrix}, \tag{2.10}$$

where, for $i = 0, \cdots, s-1, j = 0, \cdots, t-1$, $\mathbf{A}_{j,i}$'s are $N/t \times N/s$ sub-matrices of $\mathbf{A}$ and $\mathbf{B}_{i,j}$'s are $N/s \times N/t$ sub-matrices of $\mathbf{B}$. We choose s and t such that both s and t divide N and $st = m$.

***Master node (encoding)**: Define the encoding polynomials as:*

$$p_{\mathbf{A}}(x, y) = \sum_{i=0}^{t-1} \sum_{j=0}^{s-1} \mathbf{A}_{i,j} x^i y^j,$$
$$p_{\mathbf{B}}(y, z) = \sum_{k=0}^{s-1} \sum_{l=0}^{t-1} \mathbf{B}_{k,l} y^{s-1-k} z^l. \tag{2.11}$$

The master node sends the evaluations of $p_{\mathbf{A}}(x, y), p_{\mathbf{B}}(y, z)$ at $x = x_r, y = x_r^t, z = x_r^{t(2s-1)}$ to the r-th worker where x_r's are all distinct for $r \in \{1, 2, \ldots, P\}$. By this substitution, we are transforming the three-variable polynomial to a single-variable polynomial as follows:

$$p_{\mathbf{C}}(x, y, z) = p_{\mathbf{C}}(x) = \sum_{i,j,k,l} \mathbf{A}_{i,j} \mathbf{B}_{k,l} x^{i+t(s-1+j-k)+t(2s-1)l},$$

and evaluate the polynomial $p_{\mathbf{C}}(x)$ at x_r for $r = 1, \cdots, P$. In Lemma 2.1.1, we show that this transformation is one-to-one.

***Worker nodes**: For $r \in \{1, 2, \ldots, P\}$, the r-th worker node computes the matrix product $p_{\mathbf{C}}(x_r, y_r, z_r) = p_{\mathbf{A}}(x_r, y_r) p_{\mathbf{B}}(y_r, z_r)$ and sends it to the fusion node on successful completion.*

***Fusion node (decoding)**: The fusion node uses outputs of the first $t^2(2s-1)$ successful workers to compute the coefficient of $x^{i-1} y^{s-1} z^{l-1}$ in $p_{\mathbf{C}}(x, y, z) = p_{\mathbf{A}}(x, y) p_{\mathbf{B}}(y, z)$, i.e., it*

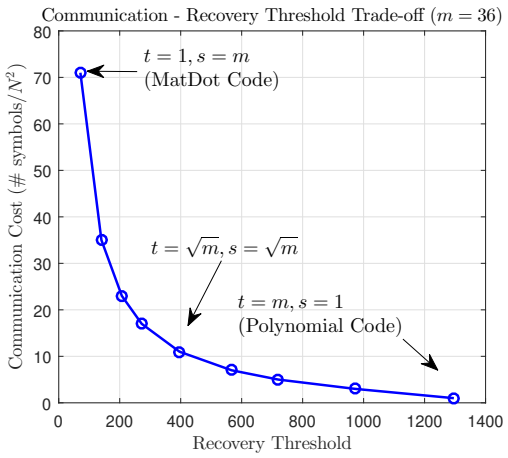


Figure 2.4: An illustration of the trade-off between communication cost (from the workers to the fusion node) and the recovery threshold of PolyDot codes by varying s and t for a fixed m ($m = 36$). The minimum communication cost is N^2, corresponding to polynomial codes, that have the largest recovery threshold. It is important to note here that in the above, we are *only including the communication cost from the workers to the fusion node*. The communication from the master node to the workers is not included, and it can dominate in situations when the workers are highly unreliable.

computes the coefficient of $x^{i-1+(s-1)t+(2s-1)t(l-1)}$ *of the transformed single-variable polynomial. The proof of Theorem 2.1.2 shows that this is indeed possible. If the number of successful workers is smaller than* $t^2(2s-1)$*, the fusion node declares a failure.*

Discussion on applicability of PolyDot codes: Before we prove the theorem, let us discuss the utility of PolyDot codes. Under a fixed storage constraint ($1/m$), as t increases and s decreases while keeping $st(= m)$ fixed, the recovery threshold keeps increasing and the computation and communication costs keep decreasing. By choosing different s and t, we can trade off communication/computation cost and recovery threshold. For $s = m$ and $t = 1$, PolyDot$(m, s = m, t = 1)$ code is a MatDot code which has a low recovery threshold but a high communication/computation cost. At the other extreme, for $s = 1$ and $t = m$, PolyDot$(m, s = 1, t = m)$ code is a Polynomial code. Now, let us consider a code with intermediate s and t values, such

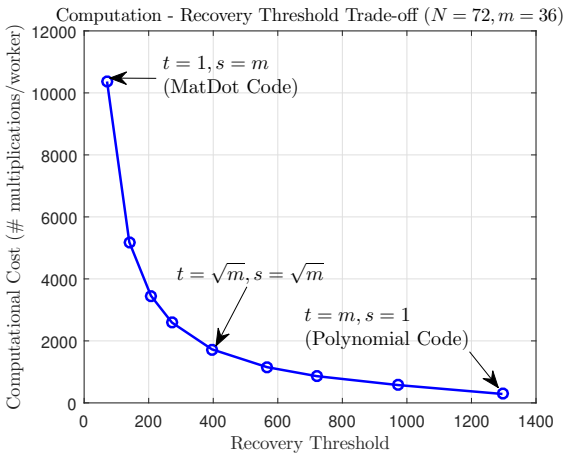


Figure 2.5: An illustration of the trade-off between the computation cost per worker and the recovery threshold of PolyDot codes by varying s and t for a fixed N, m ($N = 72, m = 36$). The minimum computation cost per worker is 288 multiplication operations per worker, corresponding to polynomial codes, that have the largest recovery threshold.

as, $s = \sqrt{m}$ and $t = \sqrt{m}$. A PolyDot$(m, s = \sqrt{m}, t = \sqrt{m})$ code has a recovery threshold of $m(2\sqrt{m} - 1) = \Theta(m^{1.5})$, and the total number of symbols to be communicated to the fusion node is $\Theta\left((N/\sqrt{m})^2 \cdot m^{1.5}\right) = \Theta(\sqrt{m}N^2)$, which is smaller than $\Theta(mN^2)$ as required by MatDot codes but larger than $\Theta(N^2)$ as required by Polynomial codes. This trade-off between communication cost and recovery threshold is illustrated in Fig. 2.4 for $m = 36$. Similarly, in terms of computational cost per worker node, a PolyDot$(m, s = \sqrt{m}, t = \sqrt{m})$ code requires $\mathcal{O}(N^3/m^{1.5})$ operations, which is less than the $\mathcal{O}(N^3/m)$ operations required by MatDot codes but higher than the $\mathcal{O}(N^3/m^2)$ operations required by Polynomial codes. This trade-off between the computation per worker and the recovery threshold is illustrated in Fig. 2.5 for $N = 72, m = 36$.

In regimes where the storage-constraint is more critical than the computation or communication time, PolyDot codes with the MatDot configuration (or at least closer to MatDot codes, *i.e.*, higher s, lower t) is more appropriate. Alternatively, in settings where computation and communication time dominate significantly, PolyDot codes with Polynomial codes' configuration (or at

least close to Polynomial codes, *i.e.*, higher t, lower s) may be more preferable. Interestingly though, even in systems where communication costs may be significant, it is possible that more communication from fewer successful workers is less expensive than requiring more successful workers as required in Polynomial codes, which we hope to explore experimentally in future work.

Now, we proceed to prove Theorem 2.1.2. We need the following lemma.

Lemma 2.1.1. *The following function*

$$\begin{aligned} f :&\{0, \cdots, t-1\} \times \{0, \cdots, 2s-2\} \times \{0, \cdots, t-1\} \\ &\to \{0, \cdots, t^2(2s-1)-1\} \\ &(\alpha, \beta, \gamma) \mapsto \alpha + t\beta + t(2s-1)\gamma \end{aligned} \tag{2.12}$$

is a bijection.

Proof. Let us assume, for the sake of contradiction, that for some $(\alpha', \beta', \gamma') \neq (\alpha, \beta, \gamma)$, $f(\alpha', \beta', \gamma') = f(\alpha, \beta, \gamma)$. Then $(f(\alpha, \beta, \gamma) \bmod t) = \alpha = (f(\alpha', \beta', \gamma') \bmod t) = \alpha'$ and hence $\alpha = \alpha'$. Similarly, $(f(\alpha, \beta, \gamma) \bmod t(2s-1)) = (f(\alpha', \beta', \gamma') \bmod t(2s-1))$ gives $\alpha + t\beta = \alpha' + t\beta'$, and thus $\beta = \beta'$ (because $\alpha = \alpha'$). Now, because $\alpha = \alpha'$ and $\beta = \beta'$, as we just established, $f(\alpha, \beta, \gamma) = f(\alpha', \beta', \gamma')$ from our assumption, it follows that $\gamma = \gamma'$. This contradicts our assumption that $(\alpha, \beta, \gamma) \neq (\alpha', \beta', \gamma')$. ■

Proof of Theorem 2.1.2. The product of $p_{\mathbf{A}}(x, y)$ and $p_{\mathbf{B}}(y, z)$ can be written as follows:

$$\begin{aligned} p_{\mathbf{C}}(x, y, z) &= p_{\mathbf{A}}(x, y) p_{\mathbf{B}}(y, z) \\ &= \left(\sum_{i=0}^{t-1} \sum_{j=0}^{s-1} \mathbf{A}_{i,j} x^i y^j \right) \left(\sum_{k=0}^{s-1} \sum_{l=0}^{t-1} \mathbf{B}_{k,l} y^{s-1-k} z^l \right) \\ &= \sum_{i,j,k,l} \mathbf{A}_{i,j} \mathbf{B}_{k,l} x^i y^{s-1+j-k} z^l. \end{aligned} \tag{2.13}$$

Note that the coefficient of $x^{i-1} y^{s-1} z^{l-1}$ in $p_{\mathbf{C}}(x, y, z)$ is equal to $\mathbf{C}_{i,l} = \sum_{k=0}^{s-1} \mathbf{A}_{i,k} \mathbf{B}_{k,l}$. By

our choice of $y = x^t$ and $z = x^{t(2s-1)}$ we can further simplify $p_{\mathbf{C}}(x, x^t, x^{t(2s-1)})$:

$$p_{\mathbf{C}}(x, y, z) = p_{\mathbf{C}}(x) = \sum_{i,j,k,l} \mathbf{A}_{i,j}\mathbf{B}_{k,l}x^{i+t(s-1+j-k)+t(2s-1)l}. \tag{2.14}$$

The maximum degree of this polynomial is when $i = t-1, j-k = s-1$ and $l = t-1$, which is $(t-1) + (2s-2)t + t(2s-1)(t-1) = t^2(2s-1) - 1$. Furthermore, if we let $\alpha = i, \beta = s-1+j-k, \gamma = l$, the function $f(\alpha, \beta, \gamma)$ in Lemma 2.1.1 is the degree of x in (2.14). This implies that for different pairs of $(i, j-k, l)$, we get different powers of x. When $j - k = 0$, we obtain $(\sum_{k=0}^{s-1} \mathbf{A}_{i,k}\mathbf{B}_{k,l})x^{i+t(s-1)+t(2s-1)l} = \mathbf{C}_{i,l}x^{i+t(s-1)+t(2s-1)l}$ which is the desired product we want to recover.

This implies that if we have $t^2(2s-1)$ successful worker nodes, we can compute all the coefficients in (2.14) by polynomial interpolation. Hence, we can recover all $\mathbf{C}_{i,l}$'s, i.e., the coefficients of $x^{i+t(s-1)+t(2s-1)l}$, for $i, l = 0, \cdots, t-1$. ■

Remark 2.1.3. We first introduce the novel PolyDot framework for *matrix-matrix* multiplication which block-partitions the two matrices $\mathbf{A}$ and $\mathbf{B}$ into $t \times s$ and $s \times t$ respectively, using two multivariate polynomials:

$$\begin{aligned} p_{\mathbf{A}}(x, y) &= \sum_{i=0}^{t-1}\sum_{j=0}^{s-1} \mathbf{A}_{i,j}x^i y^j, \\ p_{\mathbf{B}}(y, z) &= \sum_{k=0}^{s-1}\sum_{l=0}^{t-1} \mathbf{B}_{k,l}y^{s-1-k}z^l. \end{aligned} \tag{2.15}$$

It is trivial to see that for an asymmetric partitioning, e.g., where $\mathbf{A}$ is split in $t_1 \times s$ and $\mathbf{B}$ is split in $s \times t_2$ blocks, the encoding polynomials in the PolyDot framework change as:

$$\begin{aligned} p_{\mathbf{A}}(x, y) &= \sum_{i=0}^{t_1-1}\sum_{j=0}^{s-1} \mathbf{A}_{i,j}x^i y^j, \\ p_{\mathbf{B}}(y, z) &= \sum_{k=0}^{s-1}\sum_{l=0}^{t_2-1} \mathbf{B}_{k,l}y^{s-1-k}z^l. \end{aligned} \tag{2.16}$$

In this work, the novelty lies in cleverly choosing $p_{\mathbf{A}}(x, y)$ and $p_{\mathbf{B}}(y, z)$, such that, in the product of the two multivariate polynomials, i.e., in $p_{\mathbf{C}}(x, y, z)(= p_{\mathbf{A}}(x, y)p_{\mathbf{B}}(y, z))$ some coefficients

correspond to parts of the required resultant matrix $\mathbf{AB}$. After this, we convert the multivariate polynomial $p_{\mathbf{C}}(x, y, z)$ into a polynomial of a single variable using a substitution which *preserves bijection between all the coefficients (including the ones that are not required)*.

Because only some of the coefficients of $p_{\mathbf{C}}(x, y, z)$ are actually required for reconstructing $\mathbf{AB}$, it is not necessary to preserve bijection between all the coefficients in the polynomial of a single variable. In subsequent works [26, 132] a lower recovery threshold is obtained by choosing an improved substitution such that some of the garbage coefficients in $p_{\mathbf{C}}(x, y, z)$ align with each other resulting in a polynomial of a single variable with fewer coefficients.

2.1.2.7 Complexity Analysis of PolyDot codes

Encoding/decoding complexity: Encoding for one worker requires the evaluation of the polynomials $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ at a unique value of x. As both the polynomials have m non-zero coefficients which are sub-matrices of $\mathbf{A}$ and $\mathbf{B}$ respectively, the encoder scales the m sub-matrices with N^2/m elements each and adds them up. This requires computational complexity of $\mathcal{O}(m \cdot N^2/m) = \mathcal{O}(N^2)$. Thus, the overall computational complexity of encoding for P worker nodes is $\mathcal{O}(N^2 P)$. One could alternatively also use fast polynomial evaluation algorithms [63, 65] to evaluate the two polynomials of respective degrees $st - 1$ and $t^2(2s - 1) - st$ at P arbitrary points, leading to an encoding complexity of at most $\mathcal{O}\left(N^2 P \frac{\log^2(st^2) \log\log(st^2)}{m}\right)$, that can be rewritten as $\mathcal{O}\left(N^2 P \frac{\log^2(m^2/s) \log\log(m^2/s)}{m}\right)$ using $st = m$.

Decoding requires interpolating t^2 coefficients of the polynomial $p_{\mathbf{C}}(x)$ of degree $t^2(2s-1)-1$ where each coefficient is of size N^2/t^2. We examine a choice of two decoding algorithms here, and interestingly, again observe a trade-off between MatDot and Polynomial codes in decoding. If we use a decoding technique similar to MatDot codes by considering the problem of deriving the required t^2 linear combinations from the inverse of the $k \times k$ Vandermonde matrix $\mathbf{V}$ and then combining the k evaluated sub-matrices sent by the worker nodes using these t^2 linear combinations, then the overall decoding complexity is $\mathcal{O}(t^2 \cdot \frac{N^2}{t^2} k + k^3) = \mathcal{O}(N^2 k + k^3)$ where

$k = t^2(2s-1)$. Again, as $k \ll N$, the complexity is dominated by the term N^2k.

Alternatively, the decoder could also choose to solve for all the coefficients of $p_{\mathbf{C}}(x)$ from the evaluations, as a single interpolation problem. There exist fast polynomial interpolation methods that have a complexity of $\mathcal{O}(k\log^2 k)$ theoretically [65] (or more practically $\mathcal{O}(k\log^2 k \log\log k)$ [63]) for a polynomial of degree $k-1$. For this problem $k = t^2(2s-1)$. Therefore, using these fast polynomial interpolation algorithms, the decoding complexity *per coefficient matrix element is* $\mathcal{O}(t^2(2s-1)\log^2 t^2(2s-1)\log\log t^2(2s-1)) = \mathcal{O}(t^2 s\log^2(m^2/s)\log\log m^2/s)$ using $m = st$. As the interpolation is performed N^2/t^2 times for the coefficient matrices of size N^2/t^2, the overall decoding complexity is $\mathcal{O}(N^2 s\log^2(m^2/s)\log\log(m^2/s))$.

Remark 2.1.4. Note that, when we substitute $t = 1, s = m$ in the second expression of decoding complexity for PolyDot codes, we get $\mathcal{O}(N^2 m\log^2(m)\log\log(m))$ which differs from the decoding complexity of MatDot and systematic MatDot codes by a factor of $\log^2(m)\log\log(m)$ although it matches with the decoding complexity of Polynomial codes for $t = m, s = 1$. This is because for MatDot codes, we only require one coefficient of the polynomial $p_{\mathbf{C}}(x)$ and hence the decoding complexity can be lowered by $\log^2(m)\log\log(m)$ by treating the matrix-inversion and the final coefficient computation separately than solving them together as a single interpolation problem as done in the second case because interpolation also produces all the other coefficients that are not required in MatDot codes. Alternatively, for Polynomial codes, it makes sense to solve a single interpolation problem as all the coefficients of $p_{\mathbf{C}}(x)$ are useful. For a general PolyDot coding scheme, one can choose to invert first and then compute only the required coefficients (first decoding algorithm) or to decode as a single interpolation problem (second decoding algorithm) depending on whether $\mathcal{O}(N^2st^2)$ or $\mathcal{O}(N^2 s\log^2(m^2/s)\log\log(m^2/s))$ is lower.

Each worker's computational complexity: Multiplication of matrices of size $N/t \times N/s$ and $N/s \times N/t$ requires $\mathcal{O}(\frac{N^3}{st^2}) = \mathcal{O}(\frac{N^3 s}{m^2})$ computations. For the decoding complexity to be negligible in comparison to the per-node computational complexity, we need either $m^2t^2 = m^4/s^2 = o(N)$ or $m^2\log^2(m^2/s)\log\log(m^2/s) = o(N)$. Similarly, for the encoding complexity

to be negligible in comparison to the per-node computational complexity, we need $m^2P/s = o(N)$.

Communication complexity: Master node communicates $\mathcal{O}(N^2/ts) = \mathcal{O}(N^2/m)$ symbols to each worker, hence total outgoing symbols from the master node will be $\mathcal{O}(PN^2/M)$. For decoding, each node sends $\mathcal{O}(N^2/t^2)$ symbols to the fusion node and the recovery threshold is $\mathcal{O}(t^2(2s-1))$. Total number of symbols communicated to the fusion node is $\mathcal{O}((2s-1)N^2)$.

2.2 Beyond Coded Matrix Multiplication: Coded Dwarfs

Hundreds of papers on ML are being published on arxiv every week, and new computation algorithms are being released at a dazzling speed. It is impossible to develop a coding technique for each and every algorithm that comes out. A sensible goal is to identify a set if computations that are small enough to be universal but also big enough to justify the overhead of coding. Matrix multiplication is certainly one indispensable computation building block in modern-day computing, including machine learning, data analytics, and scientific computing. What are other computation primitives that make up today's computing applications? In mid 2000s, the high-performance computing community arrived at a set of such canonical computations, so-called *"seven dwarfs of computation"* [19], and the set was later expanded to "thirteen dwarfs" [3]. Each dwarf is a class of computations that share similar computation and communication patterns. Below is the list of *seven computation dwarfs*:

- Dense Linear Algebra
- Sparse Linear Algebra
- Spectral Methods
- N-Body Methods
- Structured Grids
- Unstructured Grids

- Monte Carlo (MapReduce)

The seven dwarfs have served as a guideline for building and testing a parallel system [15, 88]. We believe that looking at computation dwarfs is a good way to look at the landscape of coded computing as well. In fact, most existing works in coded computing literature belong to one of the dwarf categories, which we will call *"coded dwarfs"*. In this chapter, we want to provide a brief review of *coded dwarfs* and our perspective on the future directions of coded computing.

Dense Linear Algebra

Dense linear algebra comprises a large set of operations on dense vectors or matrices which is large classified into three categories: vector-vector operations, matrix-vector operations, and matrix-matrix operations. As these operations are essential in scientific computing and machine learning [20], some of the first analog "nanofunctions" have been built to support them [80, 118]. At system-level, there exist multiple libraries implementing these operations (e.g., BLAS and LAPACK). It is also one of the dwarfs that are substantially studied.

For matrix-vector multiplication, a recent work [67] proposed the use of Minimum Distance Separable (MDS) codes 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 [52]. Specifically for short and fat linear transforms, which is commonly used in processing high-dimensional data such as principal component analysis (PCA), short-dot codes [27] were proposed. Short-Dot codes trade off between the length of the dot products s and the recovery threshold $K = P - \frac{Ps}{N} + M$. The MDS coding strategy and the uncoded strategy are two special cases of Short-Dot codes.

For matrix-matrix multiplication, there have been an ample amount of research including: ABFT/Product codes [52, 67], high-dimensional product codes [66], Polynomial codes [130], MatDot codes [29], PolyDot and Generalized PolyDot codes [28, 132]. The performance com-

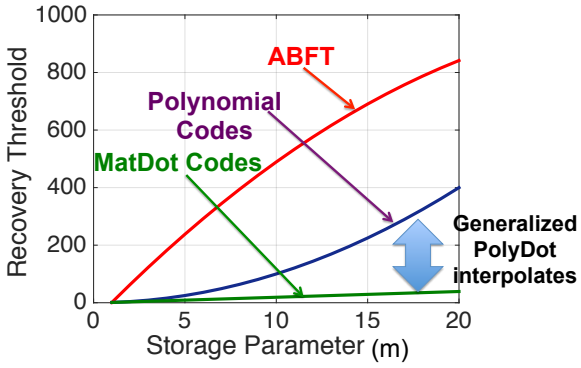


Figure 2.6: Scaling of recovery threshold with storage parameter m, *i.e.*, when each node can store a fraction $1/m$ of each of the matrices being multiplied. Total number of nodes is $P = 1000$. MatDot codes achieve the lowest recovery threshold for the storage constrained matrix multiplication problem. Generalized PolyDot codes interpolate between MatDot codes and Polynomial codes. (Figure from [28])

parison of these is depicted in Figure 2.6. we refer the reader to the previous chapter.

Another interesting line of work is coded binary linear transform with entirely unreliable components [124]. This is a departure from a common assumption in coded computing that the computing engines are unreliable, but that the encoding/decoding mechanisms can be performed reliably. In [124], "ENcoded COmputation with Decoders EmbeddeD," (or "ENCODED") was proposed where embedded decoding units to combat information dissipation [31] that makes errors to accumulate over the computation paths.

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 datasets for machine learning applications are very sparse (e.g., user ratings on products). How-

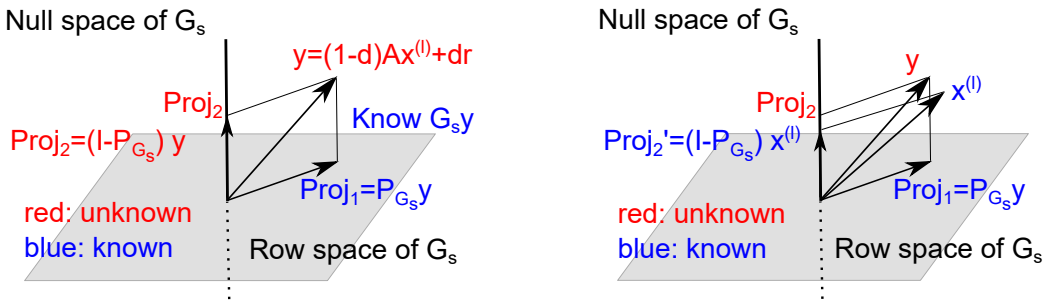


Figure 2.7: Key ideas of *substitute decoding*. (a) Finding the maximal information of the vector $\mathbf{y}$ by projecting it onto the row space of $\mathbf{G}_s$. (b) Approximating the unknown part of $\mathbf{y}$ using the result from the last iteration, $\mathbf{x}^{(l)}$.

ever, coding sparse matrices poses a fundamental challenge. Traditional codes require dense linear combinations of input data, and this will significantly increase the number of non-zero elements, taking away the storage/computation advantage of sparse data.

For sparse matrix multiplication, using sparse codes was proposed [119], but having sparsity in the generator matrix limits the fault tolerance (the number of tolerable faults is linearly proportional to the number of non-zeros in a codeword).

For solving sparse linear systems, novel ideas that exploit the iterative nature of *power iterations methods* [97] were suggested [126, 127]. Consider solving the PageRank equation [83]:

$$\mathbf{x} = (1-d)\mathbf{A}\mathbf{x} + d\mathbf{r}, \tag{2.17}$$

where the matrix $\mathbf{A}$ is extremely sparse. This can be solved through the power iterations:

$$\mathbf{x}^{(l+1)} = (1-d)\mathbf{A}\mathbf{x}^{(l)} + d\mathbf{r} \tag{2.18}$$

until $\mathbf{x}^{(l)}$ converges to the fixed point of (2.17). The novel idea of *"substitute decoding"* was proposed that utilizes the result from the previous iteration, $\mathbf{x}^{(l)}$, as a side information to recover $\mathbf{x}^{(l)}$ $\mathbf{x}^{(l+1)}$ assuming that the difference between two iterations is small. This idea is summarized in Figure 2.7.

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, which combine multiply-add operations, and employ a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others [3]. 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 [50, 96].

Several ABFT techniques for FFT [59, 82, 109, 120] were proposed for FFT circuit implementations, and more recent works studied coded FFT algorithms for for distributed FFT computation [56, 131]. This will be discussed in detail in Chapter 3.2.2.

MapReduce

MapReduce is a widely used framework in large-scale data processing. It has two phases, *"map"* and *"reduce"*. In the map phase, the 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. The *"Monte Carlo"* dwarf in the original seven dwarfs was later generalized to *MapReduce* as the pattern of communication and computing essentially follows that of Monte Carlo [3].

Coded MapReduce was suggested by Lit et al. [69] not for fault tolerance, but for reducing communication cost during the *"data shuffling"*. Between the map and 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 server. Often, this data shuffling bottlenecks the performance of MapReduce computations. Their follow-up work showed that coded MapReduce for linear operations can be used as a fault-tolerance as well [70].

The crux of the coded MapReduce strategy [69, 70] is to leverages the tradeoff between computation and communication: add redundant computations at each worker node to reduce the amount of data that has to be communicated during the shuffle stage. We illustrate this through a simple example in Figure 2.8. Some notations required are:

- N: number of input files
- K: number of compute nodes
- Q: number of output functions
- r: computation load ($1 \leqslant r \leqslant K$) – the average number of nodes that map each input file

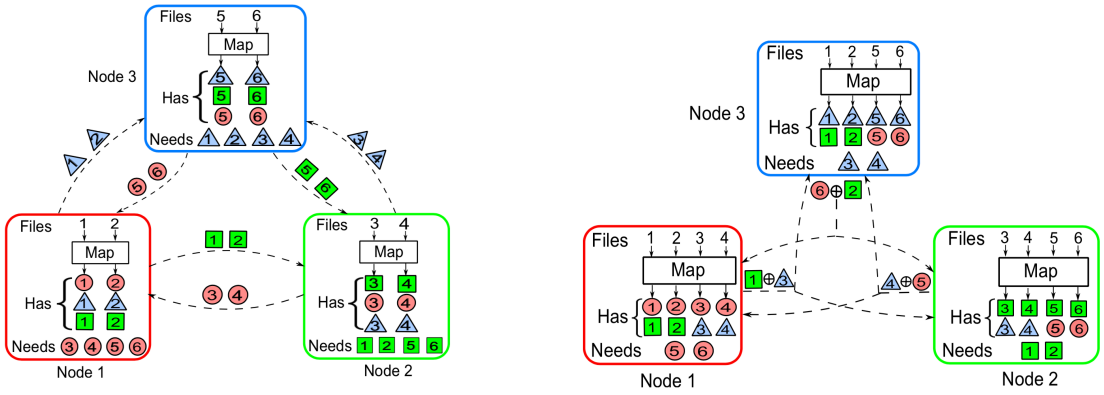
Latency gains achieved by the coded MapReduce strategy were quantified experimentally in [71].

Future Directions

Integrating these techniques closely with design of emerging devices and systems is perhaps the most important future direction. For instance, in an unpublished work with Ning Wang and Eric Pop (the authors of [118]), we developed the concept of "nanoflags." These analog engines, attached to a nanofunction, indicate the confidence a nanofunction has in its own output, based on its modeling of input dependent errors (as discussed in [118], this modeling is possible for graphene-based dot-product nanofunctions). Such novel systems that complement nanofunctions can help simplify the system-level problem because it can help identify which nodes have erroneous outputs, and discard those outputs from decoding.

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. E.g., new classes of codes were developed for flash memory [25, 43] and resistive memory [64] which are designed to combat device-specific fault patterns and vulnerabilities. Codes designed to overcome the constraints of today's distributed storage systems [24, 89, 90] 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 limitations and device-specific characteristics, newly emerging computing systems can be made robust and resilient with minimal overhead.



(a) $r = 1$ (b) $r = 2$

Figure 2.8: [71, Fig.1] An example coded MapReduce for $Q = 3, N = 6, K = 3$. Three different shapes (blue triangle, green square, red circle) represent 3 different output functions, and we use numbers to denote 6 different input files. The goal is to compute these 3 output functions on all 6 input files. During the data shuffle stage, we want to send all the values associated with the same output function to the same node – all the red circle outputs to Node 1, green square to Node 2, and blue triangle to Node 3. (a) Uncoded: Data is located in only one server, and hence $r = 1$. For the data rearrangement before the reduce phase, each node has to send two of its outputs to the other nodes. Thus, 4 intermediate values should be communicated from each node, and the total of 12 values need to be communication. (b) Coded: Each input file is present in two nodes, *i.e.,* $r = 2$. 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 sending the XOR of the intermediate values, the communication can be reduced to multicasting three values.

Chapter 3

Masterless Coded Computing

A majority of suggested coded computing strategies assume a master-worker setup where a system has a powerful master node that distributes data to and aggregates the result from worker nodes. However, this is not a practical system model especially when the scale of computation grows. When we scale the computation to thousands of worker nodes, it means that a master node must communicate with the thousands of nodes simultaneously. A similar problem called *"TCP incast problem"* is well-known in distributed storage literature [23, 93, 114]. *TCP incast* is a catastrophic TCP throughput collapse that happens when synchronized request workloads flood in past the ability of an Ethernet switch to buffer packets [87]. Similarly, large data movements to a single master node will create an unbalanced communication pattern that could result in switch buffer overflows and packet drops. Even without any issue in the network, the processing of thousands of packets at the master node will cause significant latency. We will present some experimental evidence of this in Chapter 4. Furthermore, the master-worker setup assumes that a master node has a very large memory that can store the entire data. Let us denote the number of workers as P. Then, the master node must have P times more memory than the workers, which is not realistic as P becomes large. Lastly, a master node itself can fail, at which point, computation results cannot be guaranteed. For the aforementioned reasons, large-scale parallel algorithms for HPC applications are generally designed for fully-distributed nodes without a

master. Introducing a master node just for the sake of coding would not be a compelling solution to practitioners.

In this thesis, we propose *"masterless coded computing"*. The goal of masterless coded computing is to design encoding and decoding strategies that can be performed without the presence of a master node so that they can be seamlessly integrated into existing fully-parallel algorithms.

An important challenge that has to be addressed in masterless coded computing is the communication overhead of coding. With the absence of a master node, all the nodes have to communicate with each other to perform distributed encoding and decoding. In distributed computing, communication is often the bottleneck, not computation, because communication bandwidth is not growing as fast as flop rates of processors [99]. If we blindly apply existing coded computing techniques to the fully distributed setting, the communication overhead of encoding/decoding could dominate and coded computing approach could end up much slower than the uncoded counterpart.

In Chapter 3.1, we will discuss how we can reduce the communication overhead of distributed decoding, especially in coded matrix multiplication. To the best of our knowledge, reducing communication overhead for distributed encoding is largely an open problem. However, we believe that existing works in sparse codes can be utilized [37, 48, 76, 123]. In Chapter 3.2, we will go over masterless coded computing algorithms that are designed for popularly-used parallel algorithms in practice: SUMMA for matrix multiplication and the *transpose* algorithm for FFT. In each of them, we will thoroughly analyze the communication overhead of coding and show conditions under which the overhead can be amortized.

3.1 Reducing Communication Overhead in Distributed Decoding

In this section, we introduce two ideas for reducing communication overhead of distributed decoding: locally-recoverable coded matrix multiplication and systematic coded matrix multiplication. We will explain at the beginning of each subsection how these ideas can be beneficial for reducing communication costs in masterless coded computing. Finally, we will show a code construction that is locally recoverable and systematic.

3.1.1 Locally Recoverable Coded Matrix Multiplication

3.1.1.1 Motivation

Locally recoverable (LRC) codes have been extensively studied for distributed storage as they can reduce the number of node access to repair a failed storage node [12, 41, 42, 51, 81, 84, 102, 106, 107]. In classical MDS codes, which are optimal in terms of the total amount of redundancy, we always need k symbols to recover the original message when we use an (n, k) MDS code. On the other hand, LRC codes let us to recover a lost symbol using just r other symbols where $r < k$ for the case of single erasure. Using LRC codes, when one storage node fails, we can recover it using just a few local nodes.

Having repair locality can also be useful in distributed computing for several scenarios:

- When we want to perform consecutive matrix multiplications, e.g., computing the product $\mathbf{D} = \mathbf{ABC}$, we can repair a failed node locally after computing the first product $\mathbf{D}' = \mathbf{AB}$. Then, we carry on the next computation $\mathbf{D} = \mathbf{D}'\mathbf{C}$ without all the nodes sending their intermediate results to the master node.
- In the fully distributed setting, which does not have a powerful master node, we can use a systematic code with locality. Assuming that the fault rate is low and single node failure is the most common scenario, we can recover a failed systematic node by contacting only

a few other nodes.

To the best of our knowledge, this is the first work that proposes locally recoverable matrix multiplication codes. We leverage novel matrix multiplication codes [29, 130] and optimal LRC codes [106] to obtain locally-recoverable Polynomial codes (which require minimal communication from workers to master node) and locally-recoverable MatDot codes (which are storage optimal).

3.1.1.2 Preliminaries on LRC Codes

We say that a code $\mathcal{C}$ has locality r if every symbol of the codeword can be recovered from a subset of r other symbols. In [41], a Singleton-type bound was derived on the maximum distance of LRC codes with locality r.

Theorem 3.1.1. *Let $\mathcal{C}$ be an (n, k, r) LRC code. Then the minimum distance of $\mathcal{C}$ satisfies:*

$$d \leqslant n - k - \lceil \frac{k}{r} \rceil + 2. \tag{3.1}$$

Comparing this with the (n, k) MDS code without locality which has $d = n - k + 1$, we can see that the overhead of having locality is at least $\lceil \frac{k}{r} \rceil - 1$. In this work, we use a family of optimal LRC codes presented in [106] that achieves the equality in (3.1).

Construction 3.1.1 (Optimal (n,k,r) LRC code [106])**.** *Let $\boldsymbol{a} \in \mathbb{F}_q^k$ be a message vector and let us re-index $\boldsymbol{a}$ as $\boldsymbol{a} = (a_{ij}, i = 1, \cdots, r; j = 1, \cdots \frac{k}{r})$. For simplicity, we will assume that r divides k here. Then, the encoding polynomial is defined as:*

$$f_{\boldsymbol{a}}(x) = \sum_{i=1}^{r} \sum_{j=1}^{\frac{k}{r}} a_{ij} x^{i-1} g(x)^{j-1}. \tag{3.2}$$

Let $\mathcal{A} = \{\alpha_1, \cdots, \alpha_n\}$ be a subset of $\mathbb{F}_q$ ($q \geqslant n$). The codeword is the evaluation of the polynomial $f_{\boldsymbol{a}}$ at $\alpha_1, \cdots, \alpha_n$: $\boldsymbol{c} = (f_{\boldsymbol{a}}(\alpha), \alpha \in \mathcal{A})$. A core of this construction is choosing a good polynomial $g(x)$ which satisfies the following:

i) $\deg(g) = r + 1$.

ii) There exists a partition of $\mathcal{A}$, $\mathcal{A} = \mathcal{A}_1 \bigcup \cdots \bigcup \mathcal{A}_{\frac{n}{r+1}}$, where $|\mathcal{A}_i| = r+1$ such that g is constant on each set $\mathcal{A}_i$. In other words, for all $\alpha, \alpha' \in \mathcal{A}_i$, $g(\alpha) = g(\alpha')$.

First, note that by choosing g with degree $r+1$, the degree of $f_{\boldsymbol{a}}$ becomes $(r+1)\cdot(k/r-1)+r-1 = k + r/k - 2$. Hence, the distance $d = n - k - \frac{k}{r} + 2$. This satisfies the equality in (3.1).

Now, let us see how choosing such a g guarantees locality r. Let us denote $\mathcal{A}_1 = \{\alpha_1, \cdots, \alpha_{r+1}\}$. Without loss of generality, let us assume that $f_{\boldsymbol{a}}(\alpha_1)$ is lost. We want to recover $f_{\boldsymbol{a}}(\alpha_1)$ using r other symbols. Note that, by the second condition, $g(\alpha_1) = g(\alpha_2) = \cdots = g(\alpha_{r+1}) = \gamma$. Then, $f_{\boldsymbol{a}}(x)$ at $\alpha_1, \cdots, \alpha_{r+1}$ can be represented as:

$$\begin{aligned} f_{\boldsymbol{a}}(\alpha_l) &= \sum_{i=1}^{r} \sum_{j=1}^{\frac{k}{r}} a_{ij} \alpha_l^{i-1} \gamma^{j-1} \\ &= \sum_{i=1}^{r} \big(\sum_{j=1}^{\frac{k}{r}} a_{ij} \gamma^{j-1} \big) \alpha_l^{i-1} \\ &= \sum_{i=1}^{r} \psi_i \alpha_l^{i-1} \quad (l = 1, \cdots, r+1) \end{aligned}$$

Since this is degree-$(r-1)$ polynomial in α_l, the coefficients, ψ_i's can be recovered from evaluation at r points: $f_{\boldsymbol{a}}(\alpha_2), \cdots, f_{\boldsymbol{a}}(\alpha_{r+1})$. Then, we can recover $f_{\boldsymbol{a}}(\alpha_1)$ by computing:

$$f_{\boldsymbol{a}}(\alpha_1) = \sum_{i=1}^{r} \psi_i \alpha_1^{i-1}.$$

■

3.1.1.3 Problem Statement

We want to give a coding strategy for computing $\mathbf{C} = \mathbf{AB}$ with locality r. More specifically, we want to construct locally recoverable Polynomial codes with locality r and locally recoverable MatDot codes with locality r.

3.1.1.4 Locally Recoverable Polynomial Codes

We first give an example of locally recoverable Polynomial codes for $m = 4$ and $r = 4$ with $P = 25$ worker nodes.

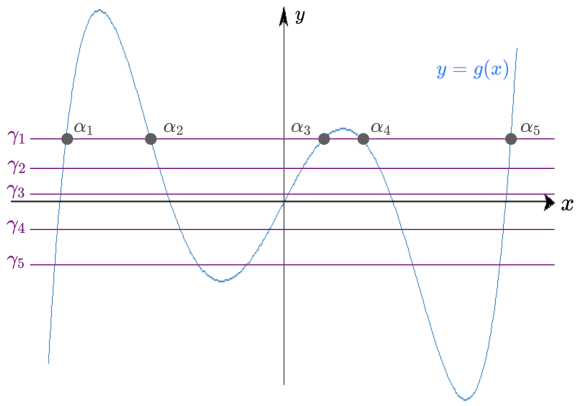


Figure 3.1: In Example 3.1.1, we have to find a degree-5 polynomial and $\gamma_1, \cdots, \gamma_5$ which satisfies $g(\mathcal{A}_i) = \gamma_i$. The plot shows one possible choice of $g(x)$ and $\gamma_1, \cdots, \gamma_5$. After choosing $g(x)$ and γ_i's ($i = 1, \cdots, 5$), α_j's are automatically decided ($j = 1, \cdots, 25$). For instance, $\mathcal{A}_1 = \{\alpha_1, \cdots, \alpha_5\}$ are shown on the plot.

Example 3.1.1 ($m = 4, r = 4, P = 25$)**.** *We first split the matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *into* 4 *blocks as follows:*

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \mathbf{A}_4 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{B}_3 & \mathbf{B}_4 \end{bmatrix}, \tag{3.3}$$

where $\mathbf{A}_i$*'s and* $\mathbf{B}_i$*'s are* $N/4 \times N$ *and* $N \times N/4$ *dimensional submatrices, respectively. Let* $\mathcal{A} = \{\alpha_1, \cdots, \alpha_{25}\}$ *be a set of* 25 *distinct real numbers and let* $\mathcal{A}_1 = \{\alpha_1, \cdots, \alpha_5\}, \cdots, \mathcal{A}_5 = \{\alpha_{21}, \cdots, \alpha_{25}\}$ *be subsets of* $\mathcal{A}$ *that form a partition of* $\mathcal{A}$*.*

Then, we encode the matrices $\mathbf{A}$ *and* $\mathbf{B}$ *with the following polynomials:*

$$p_{\mathbf{A}}(x) = \mathbf{A}_1 + \mathbf{A}_2 x + \mathbf{A}_3 x^2 + \mathbf{A}_4 x^3$$

$$p_{\mathbf{B}}(x) = \mathbf{B}_1 + \mathbf{B}_2 g(x) + \mathbf{B}_3 g(x)^2 + \mathbf{B}_4 g(x)^3$$

where $g(x)$ *is a polynomial of degree* 5 *that satisfies* $g(\mathcal{A}_i) = \gamma_i$*. An example choice of* $g(x)$ *and* γ_i*'s is shown in Fig 3.1.*

The i-th worker gets the encoded matrices, which are the evaluations of the polynomials $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ at $x = \alpha_i$. The i-th worker then computes the following product:

$$p_{\mathbf{C}}(x) = \sum_{i=1}^{4} \sum_{j=1}^{4} \mathbf{A}_i \mathbf{B}_j x^{i-1} g(x)^{j-1} \tag{3.4}$$

at $x = \alpha_i$ and returns the result to the master node.

The degree of polynomial $p_{\mathbf{C}}(x)$ is $3 \cdot 5 + 3 = 18$, so the master node can recover the coefficients of $p_{\mathbf{C}}(x)$ from its evaluation at any 19 distinct points. Hence, the recovery threshold $K = 19$.

To see that locality $r = 4$, let us assume that node 3 is erased, and notice that $g(\cdot)$ satisfies $g(\alpha_1) = g(\alpha_2) = g(\alpha_3) = g(\alpha_4) = g(\alpha_5) = \gamma_1$. Now, $p_{\mathbf{C}}(x)$ at $\alpha_1, \cdots, \alpha_5$ can be rewritten as:

$$p_{\mathbf{C}}(x) = \sum_{i=1}^{4} \left(\sum_{j=1}^{4} \mathbf{B}_j \gamma_1^{j-1} \right) \mathbf{A}_i x^{i-1}. \tag{3.5}$$

Notice that this is a polynomial of degree 3 which can be recovered from evaluation at any four distinct points, and in this case, $\alpha_1, \alpha_2, \alpha_4, \alpha_5$. ■

We now provide a general construction of LRC Polynomial codes. Note that our construction is limited to the case when $r = m$.

Construction 3.1.2 (LRC Polynomial code with $r = m$)**.** *Splitting of the matrices $\mathbf{A}$ and $\mathbf{B}$ follows Construction 3.1.6:*

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_m \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \; \mathbf{B}_2 \; \ldots \; \mathbf{B}_m \end{bmatrix}. \tag{3.6}$$

Let $\mathcal{A} = \{\alpha_1, \cdots \alpha_P\}$ be a set of P distinct real numbers and let $\{\mathcal{A}_1, \cdots, \mathcal{A}_{\frac{P}{r+1}}\}$ be subsets of $\mathcal{A}$ with size $(r+1)$ which form a partition of $\mathcal{A}$. For simplicity, we assume that $(r+1)$ divides P.

We encode matrix $\mathbf{A}$ and $\mathbf{B}$ using the following polynomials:

$$p_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_i x^{i-1}, \quad p_{\mathbf{B}}(x) = \sum_{i=1}^{m} \mathbf{B}_i g(x)^{i-1}, \tag{3.7}$$

where $g(x)$ is a polynomial of degree $(r+1)$ which is constant on each set $\mathcal{A}_i$. The i-th worker gets the evaluation of $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ at $x = \alpha_i$. Then a worker node computes the following product:

$$p_{\mathbf{C}}(x) = \sum_{i=1}^{m} \sum_{j=1}^{m} \mathbf{A}_i \mathbf{B}_j x^{i-1} g(x)^{j-1}, \tag{3.8}$$

and return the result to a master node. ■

Before proving the recovery threshold and locality property of LRC Polynomial code, we want to make an important remark on choosing $g(x)$ and $\mathcal{A}$.

Remark 3.1.1. [Finding a good polynomial $g(x)$ and a set $\mathcal{A}$] In [106], a major challenge was to find a suitable polynomial g over $\mathbb{F}_q$ while keeping q small. However, in this work we consider real numbers. In $\mathbb{R}$, as long as $g(x) = 0$ has $r+1$ distinct real roots (sufficient but not necessary condition), we can always find $\gamma_1, \cdots, \gamma_{\frac{P}{r+1}}$ and $\mathcal{A}_1, \cdots, \mathcal{A}_{\frac{P}{r+1}}$ that satisfies $g(\mathcal{A}_i) = \gamma_i$ $(i = 1, \cdots \frac{P}{r+1})$.

However, choosing evaluation points (α_i's) that satisfy the above condition can create numerical stability issues. The numerical stability issue is a persistent problem in coded computing when trying to extend the coding technique from finite field to $\mathbb{R}$ [46, 125]. This is because decoding MDS codes close to capacity often leads to matrices that have poor condition number [38]. Adding locality could worsen the problem. As the degree of g becomes large (*i.e.,* large r), the slope of the polynomial $g(\cdot)$ becomes steep very quickly. This will force us to choose α_i's that are very close to each other which can make the resulting Vandermonde matrix close to singular. How much locality effect the stability issue and how to choose a good polynomial $g(\cdot)$ and γ_i's to make the decoding as numerically stable as possible needs to be studied further. ■

The following theorem shows the recovery threshold and locality property of the proposed LRC Polynomial code construction.

Theorem 3.1.2. *LRC Polynomial code given in Construction 3.1.2 achieves locality $r = m$ and recovery threshold $K = m^2 + m - 1$. Hence, this is an optimal LRC code for locality $r = m$.*

Proof. The degree of the polynomial $p_{\mathbf{C}}$ is $(m-1) + (m+1)(m-1) = m^2 + m - 2$. Hence,

we can obtain the coefficients of $p_{\mathbf{C}}$ from evaluation at any m^2+m-1 distinct points. Because $x^{i-1}g(x)^{j-1}$ all have distinct degrees, we can decode $\mathbf{A}_i\mathbf{B}_j$ sequentially from the coefficients. First, recover $\mathbf{A}_m\mathbf{B}_m$ from the coefficient of x^{m^2+m-2}, then recover $\mathbf{A}_m\mathbf{B}_{m-1}$ from the coefficient of x^{m^2+m-2} and $\mathbf{A}_m\mathbf{B}_m$ that was already decoded, and so on. Thus, the recovery threshold $K = m^2+m-1$.

Locality $r = m$ is guaranteed as (3.8) follows the form of (3.2) in Construction 3.1.1 by setting $r = m$ and $k/r = m$. The overhead of having locality $r = m$ is $m-1$ which is $m^2/m - 1 = k/r - 1$. This shows the optimality of the LRC Polynomial code. ■

3.1.1.5 Locally Recoverable MatDot Codes

Before giving a general construction of locally recoverable MatDot codes, we want to give a simple example of LRC MatDot codes for $m = 6$ and $r = 3$.

Example 3.1.2 (LRC MatDot with $m = 6, r = 3$)**.** *First, we split the matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *as follows:*

$$\mathbf{A} = \left[\mathbf{A}_1\ \mathbf{A}_1\ \ldots\ \mathbf{A}_6\right], \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \vdots \\ \mathbf{B}_6 \end{bmatrix}.$$

Let $\mathcal{A} = \{\alpha_1, \cdots, \alpha_P\}$ *be a set of* P *distinct real numbers and let* $\mathcal{A}_1 = \{\alpha_1, \cdots, \alpha_4\}, \cdots, \mathcal{A}_{\frac{P}{4}} = \{\alpha_{P-3}, \cdots, \alpha_P\}$ *be subsets of* $\mathcal{A}$ *of size* 4 *that form a partition of* $\mathcal{A}$*. Let* $g(x)$ *be a polynomial with degree* 4 *that is constant on each subset* $\mathcal{A}_i$*. We encode matrix* $\mathbf{A}$ *and* $\mathbf{B}$ *as follows:*

$$p_{\mathbf{A}}(x) = (\mathbf{A}_1 + \mathbf{A}_2 x) + (\mathbf{A}_3 + \mathbf{A}_4 x)g(x) + (\mathbf{A}_5 + \mathbf{A}_6 x)g(x)^2,$$
$$p_{\mathbf{B}}(x) = (\mathbf{B}_6 + \mathbf{B}_5 x) + (\mathbf{B}_4 + \mathbf{B}_3 x)g(x) + (\mathbf{B}_2 + \mathbf{B}_1 x)g(x)^2.$$

The i*-th worker node receives the encoded matrices,* $p_{\mathbf{A}}(\alpha_i)$ *and* $p_{\mathbf{B}}(\alpha_i)$*, and then computes the fol-*

lowing product:

$$\begin{aligned} p_{\mathbf{C}}(x) =&(\mathbf{A}_1 + \mathbf{A}_2 x)(\mathbf{B}_6 + \mathbf{B}_5 x) + \cdots \\ &+ (\mathbf{A}_1\mathbf{B}_1 + \cdots \mathbf{A}_6\mathbf{B}_6) x g(x)^2 + \cdots \\ &+ (\mathbf{A}_5 + \mathbf{A}_6 x)(\mathbf{B}_2 + \mathbf{B}_1 x) g(x)^4 \end{aligned} \tag{3.9}$$

at $x = \alpha_i$. Notice that the coefficient of $xg(x)^2$ in (3.9) is $\mathbf{C} = \mathbf{A}_1\mathbf{B}_1 + \cdots \mathbf{A}_6\mathbf{B}_6$. The degree of polynomial $p_{\mathbf{C}}$ is $4 \cdot 4 + 2 = 18$, so we can recover the coefficients of the polynomial with evaluation at any 19 distinct points. After obtaining the coefficients of $p_{\mathbf{C}}$, the coefficients of $x^i g(x)^j$ for $i = 0, 1, 2, j = 0, \cdots, 4$ can be obtained as they all have distinct degrees. Hence, the recovery threshold $K = 19$.

To see the locality property, let us assume that node 3 is erased, and let us denote $g(\mathcal{A}_1) = \gamma$, i.e., $g(\alpha_1) = g(\alpha_2) = g(\alpha_3) = g(\alpha_4) = \gamma$. Then $p_{\mathbf{C}}(x)$ at $\alpha_1, \cdots, \alpha_4$ can be rewritten as:

$$\begin{aligned} p_{\mathbf{C}}(x) =&(\mathbf{A}_1 + \mathbf{A}_2 x)(\mathbf{B}_6 + \mathbf{B}_5 x) + \cdots \\ &+ (\mathbf{A}_1\mathbf{B}_1 + \cdots \mathbf{A}_6\mathbf{B}_6) x \gamma^2 + \cdots \\ &+ (\mathbf{A}_5 + \mathbf{A}_6 x)(\mathbf{B}_2 + \mathbf{B}_1 x) \gamma^4. \end{aligned}$$

Now, notice that this is a polynomial of degree 2, which can be recovered from evaluation at any three points, and in this case, $\alpha_1, \alpha_2, \alpha_4$. ■

We now give a construction of LRC MatDot codes with general m and r. Unlike LRC Polynomial codes, in the LRC MatDot code construction, r can take any value between 1 and $2m - 1$.

Construction 3.1.3 (LRC MatDot Codes)**.** *Splitting of the matrices $\mathbf{A}$ and $\mathbf{B}$ follows Construction 2.1.1:*

$$\mathbf{A} = \left[\mathbf{A}_1 \ \mathbf{A}_2 \ \ldots \ \mathbf{A}_m\right], \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \vdots \\ \mathbf{B}_m \end{bmatrix}. \tag{3.10}$$

Let $\mathcal{A} = \{\alpha_1, \cdots \alpha_P\}$ be a set of P distinct real numbers and let $\mathcal{A}_1, \cdots, \mathcal{A}_{\frac{P}{r+1}}$ be subsets of $\mathcal{A}$ with size $(r+1)$ which form a partition of $\mathcal{A}$. For simplicity, we assume that $r+1$ divides both P and m. Let $g(x)$ be a polynomial of degree $r+1$ which is constant on each subset $\mathcal{A}_i$. The encoding polynomials of the matrices $\mathbf{A}$ and $\mathbf{B}$ are as follows:

$$\begin{aligned} p_{\mathbf{A}}(x) &= (\mathbf{A}_1 + \cdots + \mathbf{A}_{\frac{r+1}{2}} x^{\frac{r-1}{2}}) \\ &\quad + (\mathbf{A}_{\frac{r+1}{2}+1} + \cdots + \mathbf{A}_{(r+1)} x^{\frac{r-1}{2}}) g(x) + \cdots \\ &\quad + (\mathbf{A}_{m-\frac{r+1}{2}+1} + \cdots + \mathbf{A}_m x^{(r-1)/2}) g(x)^{\frac{2m}{r+1}-1}, \qquad (3.11) \\ p_{\mathbf{B}}(x) &= (\mathbf{B}_m + \cdots + \mathbf{B}_{m-\frac{r+1}{2}+1} x^{\frac{r-1}{2}}) \\ &\quad + (\mathbf{B}_{m-\frac{r+1}{2}} + \cdots + \mathbf{B}_{m-r} x^{\frac{r-1}{2}}) g(x) + \cdots \\ &\quad + (\mathbf{B}_{\frac{r+1}{2}} + \cdots + \mathbf{B}_1 x^{\frac{r-1}{2}}) g(x)^{\frac{2m}{r+1}-1}. \qquad (3.12) \end{aligned}$$

The i-th worker gets the evaluation of $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ at $x = \alpha_i$ $(i = 1, \cdots, P)$. Then a worker node computes the following product:

$$\begin{aligned} p_{\mathbf{C}}(x) &= p_{\mathbf{A}}(x) p_{\mathbf{B}}(x) \\ &= \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_j x^{j-1} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_{m-j+1} x^{j-1} + \cdots \\ &\quad + (\mathbf{A}_1 \mathbf{B}_1 + \cdots + \mathbf{A}_m \mathbf{B}_m) x^{\frac{r-1}{2}} g(x)^{\frac{2m}{r+1}-1} + \cdots \\ &\quad + \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{m-j+1} x^{j-1} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_j x^{j-1} g(x)^{\frac{4m}{r+1}-2}, \qquad (3.13) \end{aligned}$$

and return the result to a master node. ■

The following theorem shows the locality and recovery threshold of the proposed LRC MatDot code construction.

Theorem 3.1.3. *The LRC MatDot code given in Construction 3.1.3 achieves locality r and recovery threshold $K = 4m - r - 2$.*

Proof. The degree of the polynomial $p_{\mathbf{C}}$ in (3.13) is $(r-1) + (\frac{4m}{r+1} - 2)(r+1) = 4m - r - 3$, so with evaluation at any $4m - r - 2$ distinct points, the coefficients of $p_{\mathbf{C}}$ can be recovered. Also,

notice that the coefficient of $x^{\frac{r-1}{2}}g(x)^{\frac{2m}{r+1}-1}$ is $\mathbf{C} = \mathbf{A}_1\mathbf{B}_1 + \cdots + \mathbf{A}_m\mathbf{B}_m$. As $x^i g(x)^j$ all have distinct degrees for $i = 0, \cdots \frac{r-1}{2}, j = 0, \cdots, \frac{4m}{r+1} - 2$, we can obtain the coefficient of $x^i g(x)^j$ from the coefficients of $p_{\mathbf{C}}(x)$. Hence, the recovery threshold is $K = 4m - r - 2$.

Now, let us show that the locality of the construction is r. The polynomial $p_{\mathbf{C}}(x)$ can be rewritten as:

$$\begin{aligned}
p_{\mathbf{C}}(x) &= \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_j x^{j-1} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_{m-j+1} x^{j-1} \\
&\quad + \Big(\sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_j x^{j-1} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_{m-\frac{r+1}{2}-j+1} x^{j-1} \\
&\qquad + \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{\frac{r+1}{2}+j} x^{j-1} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_{m-j+1} x^{j-1} \Big) g(x) \\
&\quad + \cdots + \Big(\sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{m-j+1} x^{j-1} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_j x^{j-1} \Big) g(x)^{\frac{4m}{r+1}-2} \\
&= f_1(x) + f_2(x)g(x) + \cdots + f_{\frac{4m}{r+1}-1}(x) g(x)^{\frac{4m}{r+1}-2}.
\end{aligned}$$

Notice that $f_1, \cdots, f_{\frac{4m}{r+1}-1}$ are all polynomials of degree $r-1$. Let $p_{\mathbf{C}}(\alpha)$ be the lost matrix and let $\alpha \in \mathcal{A}_l$. For all $\beta \in \mathcal{A}_l$,

$$p_{\mathbf{C}}(\beta) = f_1(\beta) + f_2(\beta)\gamma_l + \cdots + f_{\frac{4m}{r+1}-1}(\beta)\gamma_l^{\frac{4m}{r+1}-2}, \tag{3.14}$$

because $g(\mathcal{A}_l) = \gamma_l$. This is a degree-$(r-1)$ polynomial in β, so the coefficients of $p_{\mathbf{C}}$ can be recovered from its evaluation at the r points in $\mathcal{A}_l \backslash \{\alpha\}$. Then the lost matrix can be recovered by evaluating $p_{\mathbf{C}}(\beta)$ given in (3.14) at $\beta = \alpha$. ■

By comparing the recovery threshold given in Theorem 3.1.3 and the recovery threshold of MatDot codes without locality (Construction 2.1.1), we can see that the overhead of having locality r is $2m-r-1$. However, the optimal overhead suggested by Theorem 3.1.1 is $\lceil \frac{2m-1}{r} \rceil - 1$. Thus, there is a gap between the proposed LRC MatDot codes and the optimal LRC codes; while the optimal overhead of having locality r decreases in the order of $1/r$, the overhead of LRC

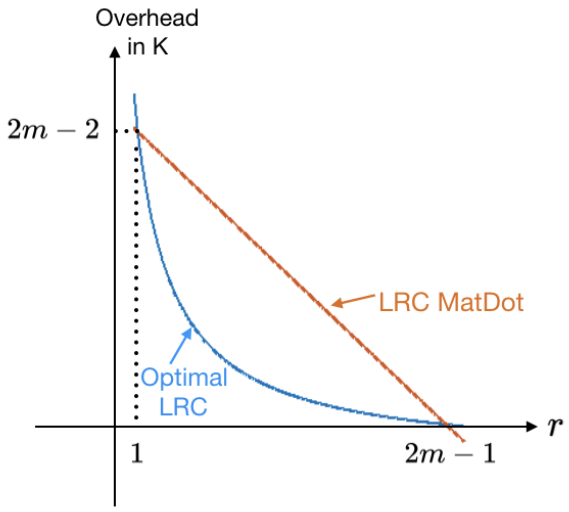


Figure 3.2: This plot shows the gap between the optimal LRC codes and the proposed LRC MatDot construction. In the optimal LRC codes, the overhead of having locality r in K is $\lceil \frac{2m-1}{r} \rceil - 1$, while in the LRC MatDot codes, the overhead is $2m - 1 - r$.

MatDot codes decreases linearly in r (see Fig 3.2). Whether this sub-optimality is inevitable due to the structure of MatDot codes is an open question.

3.1.2 Systematic Coded Matrix Multiplication

3.1.2.1 Motivation

To understand the advantages of systematic codes in a masterless setup, let us consider computing matrix product $\mathbf{C} = \mathbf{AB}$ and assume the splitting of matrices as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_m \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \cdots & \mathbf{B}_m \end{bmatrix}. \tag{3.15}$$

Note that this splitting is same as Polynomial codes [130] and Product codes [67]. In an uncoded strategy, m^2 workers will compute the product:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{A_1B_1} & \cdots & \mathbf{A_1B_m} \\ \vdots & \ddots & \vdots \\ \mathbf{A_mB_1} & \cdots & \mathbf{A_mB_m} \end{bmatrix}, \tag{3.16}$$

where each worker computes one sub-block of $\mathbf{C}$, *i.e.,* $\mathbf{A_iB_j}$. Introducing redundancy for resilience, we add p additional nodes. When failures happen during the computation, any m^2 successful nodes out of $m^2 + p$ nodes can reconstruct the computation output $\mathbf{C}$. In many settings, failures are rare. Under our masterless setting, if we use a non-systematic code, m^2 nodes have to communicate with each other to recover the product $\mathbf{C}$ even when there is no failure, which can be extremely expensive. On the other hand, if we use a systematic code, we do not need any communication to recover $\mathbf{C}$ when all m^2 systematic node – nodes that compute $\mathbf{A_iB_j}$'s $(i, j = 1, \cdots, m)$ – are successful. Further, even when there is a failure among the systematic workers, recovering a failed node only requires communication from m^2 nodes to the failed node. This has a smaller communication complexity than all m^2 nodes communicating with each other. Lastly, encoding systematic codes is more communication efficient since we only have to encode p additional nodes as compared to encoding all $m^2 + p$ nodes in non-systematic codes.

We will first give the systematic construction on MatDot codes we discussed in Section 2.1.2. Then, we discuss why it is not straightforward to construct systematic Polynomial codes. Finally, we will present a general framework for designing systematic codes for the matrix splitting given in (3.15) (as in Polynomial codes).

3.1.2.2 Systematic MatDot Codes

We will first define systematic codes in the MatDot context.

Definition 3.1.1. [Systematic code for distributed matrix multiplication problem (Section 2.1.2.2)] For the problem stated in Section 2.1.2.2 computed on the system defined in Definition 2.1.1 such

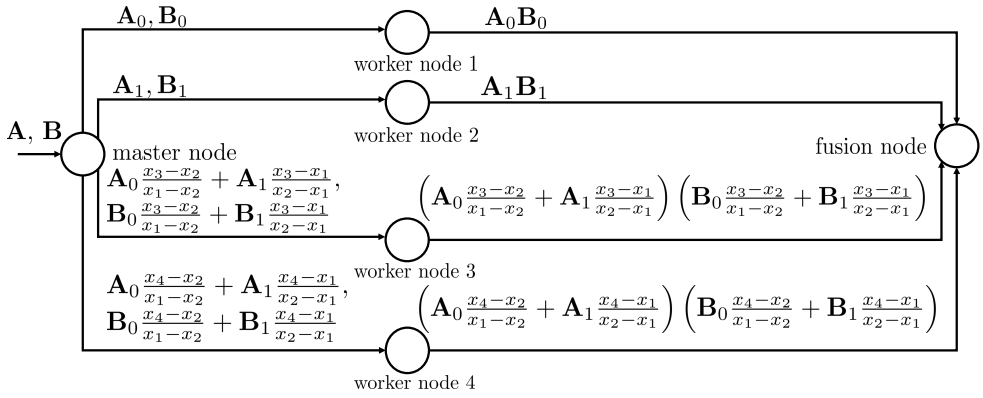


Figure 3.3: An illustration of the computational system with four worker nodes and applying systematic MatDot codes with $m = 2$. The recovery threshold is 3.

that the matrices $\mathbf{A}$ and $\mathbf{B}$ are split as in (2.3), a code is called *systematic* if the output of the r-th worker node is the product $\mathbf{A}_r\mathbf{B}_r$, for all $r \in \{1, \cdots, m\}$. We refer to the first m worker nodes, that output $\mathbf{A}_r\mathbf{B}_r$ for $r \in \{1, \cdots, m\}$, as *systematic worker nodes*.

Note that the final output $\mathbf{AB}$ can be obtained by summing up the outputs from the m systematic worker nodes:

$$\mathbf{AB} = \sum_{r=1}^{m} \mathbf{A}_{r-1}\mathbf{B}_{r-1}.$$

The presented systematic code, named *"systematic MatDot code"*, is advantageous over MatDot codes in two aspects. Firstly, even though both MatDot and systematic MatDot codes have the same recovery threshold, systematic MatDot codes can recover the output as soon as the m systematic worker nodes successfully finish, this is unlike MatDot codes which *always* require $2m - 1$ workers to successfully finish to recover the final result. Furthermore, when the m systematic worker nodes successfully finish first, the decoding complexity using systematic MatDot codes is $O(mN^2)$, which is slightly less than the decoding complexity of MatDot codes, i.e., $O(kN^2 + k^3)$ where $k = 2m - 1$. Another advantage for systematic MatDot codes over MatDot codes is that the systematic MatDot approach may be useful for *backward-compatibility with current practice*. What this means is that, for systems that are already established and operating with no straggler tolerance, but do an m-way parallelization, it is easier to apply the systematic

approach as the infrastructure could be appended to additional worker nodes without modifying what the first m nodes are doing.

The following theorem shows that there exists a systematic MatDot code construction that achieves the same recovery threshold as MatDot codes.

Theorem 3.1.4. *For the matrix-matrix multiplication problem specified in Section 2.1.2.2 computed on the system defined in Definition 2.1.1, there exists a systematic code, where the product* $\mathbf{AB}$ *is the summation of the output of the first* m *worker nodes, that solves this problem with a recovery threshold of* $2m-1$*, where* $m \geqslant 2$ *is any positive integer that divides* N*.*

Before we describe the construction of systematic MatDot codes, that will be used to prove Theorem 3.1.4, we first present a simple example to illustrate the idea of systematic MatDot codes.

Example 3.1.3. *[Systematic MatDot code,* $m = 2, k = 3$*]*

Matrix $\mathbf{A}$ *is split vertically into two sub-matrices (column-blocks)* $\mathbf{A}_0$ *and* $\mathbf{A}_1$*, each of dimension* $N \times \frac{N}{2}$ *and matrix* $\mathbf{B}$ *is split horizontally into two sub-matrices (row-blocks)* $\mathbf{B}_0$ *and* $\mathbf{B}_1$*, each of dimension* $\frac{N}{2} \times N$ *as follows:*

$$\mathbf{A} = \left[\begin{array}{cc} \mathbf{A}_0 & \mathbf{A}_1 \end{array} \right], \quad \mathbf{B} = \left[\begin{array}{c} \mathbf{B}_0 \\ \\ \mathbf{B}_1 \end{array} \right]. \tag{3.17}$$

Now, we define the encoding functions $p_{\mathbf{A}}(x)$ *and* $p_{\mathbf{B}}(x)$ *as* $p_{\mathbf{A}}(x) = \mathbf{A}_0 \frac{x-x_2}{x_1-x_2} + \mathbf{A}_1 \frac{x-x_1}{x_2-x_1}$ *and* $p_{\mathbf{B}}(x) = \mathbf{B}_0 \frac{x-x_2}{x_1-x_2} + \mathbf{B}_1 \frac{x-x_1}{x_2-x_1}$*, for distinct* $x_1, x_2 \in \mathbb{F}$*. Let* $x_3, \cdots, x_P$ *be elements of* $\mathbb{F}$ *such that* $x_1, x_2, x_3, \cdots, x_P$ *are distinct. The master node sends* $p_{\mathbf{A}}(x_r)$ *and* $p_{\mathbf{B}}(x_r)$ *to the* r*-th worker node, for all* $r \in \{1, \cdots, P\}$*, where the* r*-th worker node performs the multiplication* $p_{\mathbf{A}}(x_r)p_{\mathbf{B}}(x_r)$ *and sends the output to the fusion node. The exact computations at each worker node are depicted in Fig. 3.3.*

We can observe that the outputs of the worker nodes $1, 2$ *are* $\mathbf{A}_0\mathbf{B}_0, \mathbf{A}_1\mathbf{B}_1$*, respectively, and hence this code is systematic. Let us consider a scenario where the systematic worker nodes, i.e., worker nodes* 1 *and* 2*, complete their computations first. In this scenario, the fusion node does*

not require a decoding step and can obtain the product $\mathbf{AB}$ *by simply performing the summation of the two outputs it has received:* $\mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_1$. *Now, let us consider a different scenario where worker nodes* $1, 3, 4$ *are the first three successful workers. Then, the fusion node receives three matrices,* $p_{\mathbf{A}}(x_1)p_{\mathbf{B}}(x_1), p_{\mathbf{A}}(x_3)p_{\mathbf{B}}(x_3)$, *and* $p_{\mathbf{A}}(x_4)p_{\mathbf{B}}(x_4)$. *Since these three matrices can be seen as three evaluations of the polynomial* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *of degree* 2 *at three distinct evaluation points* x_1, x_3, x_4, *the coefficients of the polynomial* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *can be obtained using polynomial interpolation. Finally, to obtain the product* $\mathbf{AB}$, *we evaluate* $p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ *at* $x = x_1, x_2$ *and sum them up:*

$$p_{\mathbf{A}}(x_1)p_{\mathbf{B}}(x_1) + p_{\mathbf{A}}(x_2)p_{\mathbf{B}}(x_2) = \mathbf{A}_0\mathbf{B}_0 + \mathbf{A}_1\mathbf{B}_1 = \mathbf{AB}.$$

■

We now describe the general construction of the systematic MatDot codes for matrix-matrix multiplication. As all the code constructions in this paper follow the polynomial format given in Construction 2.1.1, in our subsequent constructions, we will only highlight major differences, such as, encoding polynomials.

Construction 3.1.4. *[Systematic MatDot codes]*

Splitting of input matrices*:* $\mathbf{A}$ *and* $\mathbf{B}$ *are split as in (2.3).*

Master node (encoding)*: The master node encodes matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *using the following polynomials:*

$$p_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_{i-1}L_i(x), \quad p_{\mathbf{B}}(x) = \sum_{i=1}^{m} \mathbf{B}_{i-1}L_i(x), \tag{3.18}$$

where

$$L_i(x) = \prod_{j\in\{1,\cdots,m\}\setminus\{i\}} \frac{x - x_j}{x_i - x_j}. \tag{3.19}$$

Fusion node (decoding)*: For any* k *such that* $m \leqslant k \leqslant 2m-1$, *whenever the outputs of the first* k *successful workers contain the outputs of the systematic worker nodes* $1, \cdots, m$, *i.e.,* $\{p_{\mathbf{C}}(x_r)\}_{r\in\{1,\cdots,m\}}$ *is contained in the set of the first* k *outputs received by the fusion node, the fusion node performs the summation* $\sum_{r=1}^{m} p_{\mathbf{C}}(x_r)$. *Otherwise, if* $\{p_{\mathbf{C}}(x_r)\}_{r\in\{1,\cdots,m\}}$ *is not*

contained in the set of the first $2m-1$ evaluations received by the fusion node, the fusion node performs the following steps: (i) interpolates the polynomial $p_{\mathbf{C}}(x) = p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ (the feasibility of this step will be shown later in the proof of Theorem 3.1.4), (ii) evaluates $p_{\mathbf{C}}(x)$ at $x_1, \cdots, x_m$, (iii) performs the summation $\sum_{r=1}^{m} p_{\mathbf{C}}(x_r)$.

If the number of successful worker nodes is smaller than $2m-1$ and the first m worker nodes are not included in the successful worker nodes, the fusion node declares a failure.

The following lemma proves that the construction given here is systematic.

Lemma 3.1.1. *For Construction 3.1.4, the output of the r-th worker node, for $r \in \{1, \cdots, m\}$, is the product $\mathbf{A}_{r-1}\mathbf{B}_{r-1}$. That is, Construction 3.1.4 is a systematic code for distributed matrix-matrix multiplication as defined in Definition 3.1.1*

Proof of Lemma 3.1.1. The lemma follows from the fact that $p_{\mathbf{A}}(x_r) = \mathbf{A}_{r-1}$, and $p_{\mathbf{B}}(x_r) = \mathbf{B}_{r-1}$, for $r \in \{1, \cdots, m\}$. Thus, $p_{\mathbf{C}}(x_r) = p_{\mathbf{A}}(x_r)p_{\mathbf{B}}(x_r) = \mathbf{A}_{r-1}\mathbf{B}_{r-1}$, for any $r \in \{1, \cdots, m\}$. ■

Now, we proceed with the proof of Theorem 3.1.4.

Proof of Theorem 3.1.4. Since Construction 3.1.4 is a systematic code for matrix-matrix multiplication (Lemma 3.1.1), in order to prove the theorem, it suffices to show that Construction 3.1.4 is a valid construction with a recovery threshold $k = 2m-1$. From (3.19), observe that the polynomials $L_i(x)$, $i \in \{1, \cdots, m\}$, have degrees $m-1$ each. Therefore, each of $p_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_{i-1}L_i(x)$ and $p_{\mathbf{B}}(x) = \sum_{i=1}^{m} \mathbf{B}_{i-1}L_i(x)$ has a degree of $m-1$ as well. Consequently, $p_{\mathbf{C}}(x) = p_{\mathbf{A}}(x)p_{\mathbf{B}}(x)$ has a degree of $2m-2$. Now, because the polynomial $p_{\mathbf{C}}(x)$ has degree $2m-2$, evaluation of the polynomial at any $2m-1$ distinct points is sufficient to interpolate $\mathbf{C}(x)$ using polynomial interpolation algorithm. Now, since Construction 3.1.4 is systematic (Lemma 3.1.1), the product $\mathbf{AB}$ is the summation of the outputs of the first m workers, i.e., $\mathbf{AB} = \sum_{r=1}^{m} p_{\mathbf{C}}(x_r)$. Therefore, after the fusion node interpolates $\mathbf{C}(x)$, evaluating $p_{\mathbf{C}}(x)$ at $x_1, \cdots, x_m$, and performing the summation $\sum_{r=1}^{m} p_{\mathbf{C}}(x_r)$ yields the product $\mathbf{AB}$. ■

3.1.2.2.1 Complexity Analysis of Systematic MatDot codes Apart from the encoding/decoding complexity, the complexity analyses of systematic MatDot codes are the same as their MatDot counterpart. In the following, we investigate the encoding/decoding complexity of Construction 3.1.4.

Encoding/Decoding Complexity: Encoding for each worker first requires performing evaluations of polynomials $L_i(x)$ for all $i \in \{1, \cdots, m\}$, with each evaluation requiring $\mathcal{O}(m)$ operations. This gives $\mathcal{O}(m^2)$ operations for all polynomial evaluations. Afterwards, two linear combinations of m sub-matrices of size N^2/m each is taken, which is of complexity $\mathcal{O}(mN^2/m) = \mathcal{O}(N^2)$. Therefore, the overall encoding complexity for *each non-systematic worker* is $\mathcal{O}(\max(N^2, m^2)) = \mathcal{O}(N^2)$ because $m \ll N$. For the systematic workers, no further encoding is required on the sub-matrices of $\mathbf{A}$ and $\mathbf{B}$. Thus, the overall computational complexity of encoding for P workers is $\mathcal{O}(N^2(P-m))$.

For decoding, two cases would arise depending on whether all the m systematic nodes finished first or not. When all the m systematic nodes finish first, the decoding is equivalent to taking the sum of the m systematic evaluations and is thus of complexity $\mathcal{O}(N^2 m)$. Alternatively, when the m systematic nodes do not finish first, the decoder waits for the first $k(= 2m-1)$ nodes to send their evaluations of $p_{\mathbf{C}}(x)$. Then it is required to interpolate the coefficients of $p_{\mathbf{C}}(x)$, evaluate it at the systematic points $x_1, x_2, \ldots, x_m$, and then take the sum of the systematic evaluations. Because we are interested in only the final sum of the systematic evaluations and not in the individual systematic evaluations or coefficient interpolations, we again consider the problem of deriving the appropriate linear combination and taking the final linear combination on the matrices separately.

Recall that $p_{\mathbf{C}}(x) = \mathbf{C}_0 + \mathbf{C}_1 x + \ldots + \mathbf{C}_{k-1} x^{k-1}$ where $k = 2m-1$ but now we are interested in computing the sum of the systematic evaluations of $p_{\mathbf{C}}(x)$ at $x_1, x_2, \ldots, x_m$. Also let $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k$ denote the $k(= 2m-1)$ unique values at which the k fastest workers evaluated the polynomial $p_{\mathbf{C}}(x)$ and $\mathbf{V}$ denote the $k \times k$ Vandermonde matrix as defined in (2.6). Recall

that,

$$\begin{bmatrix} \mathbf{C}_0 \\ \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{k-1} \end{bmatrix} = \left(\mathbf{V}^{-1} \otimes I_{N\times N}\right) \begin{bmatrix} p_{\mathbf{C}}(\tilde{x}_1) \\ p_{\mathbf{C}}(\tilde{x}_2) \\ \vdots \\ p_{\mathbf{C}}(\tilde{x}_k) \end{bmatrix}.$$

Let $\hat{\mathbf{V}}$ denote the $m \times k$ Vandermonde matrix for evaluation, consisting of increasing powers of the m systematic values $x_1, x_2, \ldots, x_m$, as follows:

$$\hat{\mathbf{V}} = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{k-1} \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_m & x_m^2 & \ldots & x_m^{k-1} \end{bmatrix}.$$

Now, the evaluation of $p_{\mathbf{C}}(x)$ at the systematic values $x_1, x_2, \ldots, x_m$ is equivalent to the following operation:

$$\left(\hat{\mathbf{V}} \otimes I_{N\times N}\right) \begin{bmatrix} \mathbf{C}_0 \\ \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{k-1} \end{bmatrix} = \left((\hat{\mathbf{V}}\mathbf{V}^{-1}) \otimes I_{N\times N}\right) \begin{bmatrix} p_{\mathbf{C}}(\tilde{x}_1) \\ p_{\mathbf{C}}(\tilde{x}_2) \\ \vdots \\ p_{\mathbf{C}}(\tilde{x}_k) \end{bmatrix}.$$

Finally, the summation of these m systematic evaluations can be written as:

$$\begin{aligned} &\left([1, 1, \ldots, 1]_{1\times m} \otimes I_{N\times N}\right) \left(\hat{\mathbf{V}} \otimes I_{N\times N}\right) \begin{bmatrix} \mathbf{C}_0 \\ \mathbf{C}_1 \\ \vdots \\ \mathbf{C}_{k-1} \end{bmatrix} \\ &= \left(([1, 1, \ldots, 1]_{1\times m}\hat{\mathbf{V}}\mathbf{V}^{-1}) \otimes I_{N\times N}\right) \begin{bmatrix} p_{\mathbf{C}}(\tilde{x}_1) \\ p_{\mathbf{C}}(\tilde{x}_2) \\ \vdots \\ p_{\mathbf{C}}(\tilde{x}_k) \end{bmatrix}. \end{aligned}$$

The decoder first computes the final row-vector $([1,1,\ldots,1]_{1\times m}\hat{\mathbf{V}}\mathbf{V}^{-1})$ (complexity is at most $\mathcal{O}(k^3)$ as it is dominated by the inversion of the matrix $\mathbf{V}$). Next, it linearly combines the k evaluations $p_{\mathbf{C}}(\tilde{x}_1), p_{\mathbf{C}}(\tilde{x}_2), \ldots, p_{\mathbf{C}}(\tilde{x}_k)$ (of size N^2 each) using the k values in the final row vector (complexity is $\mathcal{O}(N^2k)$). Thus, the total decoding complexity is $\mathcal{O}(N^2k+k^3) = \mathcal{O}(N^2k)$ when $k(=2m-1) \ll N$. This is similar to MatDot codes.

Note that, these encoding and decoding complexities may be improved further in functions of m and P in different scenarios, e.g., using alternate methods of faster evaluation, or using the outputs of the systematic nodes more efficiently during decoding if at least some of them are in the set of k fastest workers (if not all) that will be pursued as a future work. Here, we restrict ourselves to somewhat conservative estimates for our proposed strategy as our main goal is to explore dependence on N in the regime where $m, P \ll N$.

3.1.2.3 Systematic Polynomial Codes

Using the similar technique as we used in systematic MatDot codes does not easily yield a systematic construction for Polynomial codes [130]. We follow the univariate polynomial construction in [130] and assume that the matrices $\mathbf{A}$ and $\mathbf{B}$ are encoded using the polynomials $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$:

$$p_{\mathbf{A}}(x) = \sum_{d=1}^{D_A} f_d(\mathbf{A}_1, \ldots, \mathbf{A}_m)x^d, \tag{3.20}$$

and

$$p_{\mathbf{B}}(x) = \sum_{d=1}^{D_B} g_d(\mathbf{B}_1, \ldots, \mathbf{B}_m)x^d, \tag{3.21}$$

for some (possibly linear) functions $f_d(\cdot)$'s and $g_d(\cdot)$'s.

Let us assume that we use polynomial $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ and the first m^2 workers compute $\mathbf{A_i}\mathbf{B_j}$'s $(i,j = 1,\cdots,m)$. This implies the following:

$$p_{\mathbf{A}}(\alpha_n)p_{\mathbf{B}}(\alpha_n) = \mathbf{A}_i\mathbf{B}_j, \tag{3.22}$$

for $n = m \cdot (i-1) + j$. Then, ignoring constant factors, the following should be satisfied:

$$p_{\mathbf{A}}(\alpha_n) = \mathbf{A}_i, \quad p_{\mathbf{B}}(\alpha_n) = \mathbf{B}_j. \tag{3.23}$$

This imposes m^2 evaluation points on both $p_{\mathbf{A}}$ and $p_{\mathbf{B}}$. Hence, the degree of the polynomials $p_{\mathbf{A}}$ and $p_{\mathbf{B}}$ should be at least $m^2 - 1$. Their product, $p_{\mathbf{C}}(x) = p_{\mathbf{A}}(x) \cdot p_{\mathbf{B}}(x)$, thus has degree at least $2m^2 - 2$. This makes the recovery threshold $2m^2 - 1$, instead of m^2.

3.1.2.4 A General Description of Systematic Matrix Multiplication Codes

We first introduce some notations and set up a framework for systematic matrix multiplication codes under the matrix splitting speified in (3.15). We denote the "block-vectorized" version of the final matrix $\mathbf{C}$ by:

$$\text{block-vec}(\mathbf{C}) = \left[\mathbf{A}_1\mathbf{B}_1 \cdots \mathbf{A}_1\mathbf{B}_m \cdots \mathbf{A}_m\mathbf{B}_1 \cdots \mathbf{A}_m\mathbf{B}_m\right]^T. \tag{3.24}$$

Let us assume that the matrix blocks $\mathbf{A}_i$'s and $\mathbf{B}_j$'s are scalars for the ease of explanation. We will first explain how we encode input matrices $\mathbf{A}$ and $\mathbf{B}$ and then show how the product $\mathbf{C}$ is encoded as a result.

Systematic encoding matrices for $\mathbf{A}$ and $\mathbf{B}$ are written as:

$$G_{\mathbf{A}} = \begin{bmatrix} & I_{m\times m} \otimes \mathbf{1}_{m\times 1} & \\ \hline a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{p,1} & \cdots & a_{p,m} \end{bmatrix}, \tag{3.25}$$

$$G_{\mathbf{B}} = \left[\begin{array}{ccc} & \mathbf{1}_{m\times 1} \otimes I_{m\times m} & \\ \hline b_{1,1} & \cdots & b_{1,m} \\ \vdots & \ddots & \vdots \\ b_{p,1} & \cdots & b_{p,m} \end{array}\right]. \tag{3.26}$$

We will call the bottom submatrices of these matrices as $P_{\mathbf{A}}$ and $P_{\mathbf{B}}$ respectively, as they are the parity-generating parts. Assuming that $\mathbf{A}_1, \cdots, \mathbf{A}_m, \mathbf{B}_1, \cdots, \mathbf{B}_m$ are scalars, our encoding can be written as:

$$\begin{bmatrix} \widetilde{\mathbf{A}}_1 \\ \vdots \\ \widetilde{\mathbf{A}}_{m^2} \\ \widetilde{\mathbf{A}}_{m^2+1} \\ \vdots \\ \widetilde{\mathbf{A}}_{m^2+p} \end{bmatrix} = G_{\mathbf{A}} \begin{bmatrix} \mathbf{A_1} \\ \vdots \\ \mathbf{A}_m \end{bmatrix} = \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_m \\ \vdots \\ \mathbf{A}_m \\ a_{1,1}\mathbf{A}_1 + \cdots + a_{1,m}\mathbf{A}_m \\ \vdots \\ a_{p,1}\mathbf{A}_1 + \cdots + a_{p,m}\mathbf{A}_m \end{bmatrix}, \tag{3.27}$$

$$\begin{aligned} &\begin{bmatrix} \widetilde{\mathbf{B}}_1 \cdots \widetilde{\mathbf{B}}_{m^2} & \widetilde{\mathbf{B}}_{m^2+1} \cdots \widetilde{\mathbf{B}}_{m^2+p} \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 & \cdots & \mathbf{B}_m \end{bmatrix} G_{\mathbf{B}}^T \\ &= \Big[\mathbf{B}_1 \ \cdots \mathbf{B}_m \cdots \ \mathbf{B}_1 \cdots \mathbf{B}_m \ b_{1,1}\mathbf{B}_1 + \cdots + b_{1,m}\mathbf{B}_m \cdots \\ &b_{p,1}\mathbf{B}_1 + \cdots + b_{p,m}\mathbf{B}_m\Big]. \end{aligned} \tag{3.28}$$

$\widetilde{\mathbf{A}}_i$ and $\widetilde{\mathbf{B}}_i$ represent encoded data the node i receives ($i = 1, \cdots, m^2 + p$).

The final encoded product can now be written as:

$$\begin{aligned}\widetilde{\mathbf{C}} &= \text{block-vec}(\mathbf{C})G_{\mathbf{C}} \\ &= \begin{bmatrix} \mathbf{A}_1\mathbf{B}_1 \\ \mathbf{A}_1\mathbf{B}_2 \\ \vdots \\ \mathbf{A}_m\mathbf{B}_m \\ (a_{1,1}\mathbf{A}_1 + \cdots + a_{1,m}\mathbf{A}_m)(b_{1,1}\mathbf{B}_1 + \cdots + b_{1,m}\mathbf{B}_m) \\ \vdots \\ (a_{p,1}\mathbf{A}_1 + \cdots + a_{p,m}\mathbf{A}_m)(b_{p,1}\mathbf{B}_1 + \cdots + b_{p,m}\mathbf{B}_m) \end{bmatrix}\end{aligned} \tag{3.29}$$

which encodes the block-vectorized form in (3.24) using an encoding matrix of the form:

$$G_{\mathbf{C}} = \left[\begin{array}{c} I_{m^2 \times m^2} \\ \hline\\ P_{\mathbf{A}} \star P_{\mathbf{B}} \\ \\ \end{array}\right]. \tag{3.30}$$

The $\star$ denotes "row-wise Kronecker product", also known as the Khatri-Rao product [75].

Remark 3.1.2. Remember that we used a simplifying assumption that $\mathbf{A}_i$'s and $\mathbf{B}_j$'s are scalars. To extend this to actual matrices of dimension $N/m \times N$ and $N \times N/m$, we can treat $\mathbf{A}_i, \mathbf{B}_j$'s as elements in the vector space of $\mathbb{R}^{N/m \times N}$ and $\mathbb{R}^{N \times N/m}$. Then, we can think of the matrix $\begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_m \end{bmatrix}$ as an $m \times 1$ column vector with each element in $\mathbb{R}^{N/m \times N}$. In a similar fashion, the matrix $\begin{bmatrix} \mathbf{B}_1 & \cdots & \mathbf{B}_m \end{bmatrix}$ can be regarded as an $1 \times m$ row vector with each element in $\mathbb{R}^{N \times N/m}$. The

matrix product $G_{\mathbf{A}} \cdot \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_m \end{bmatrix}$ is now a matrix-vector product where the dimension of the matrix is $(m^2 + p) \times m$ and the length of the vector is m. Each element in the matrix is in the field $\mathbb{R}$ and each element in the vector is in the vector space $\mathbb{R}^{N/m \times N}$. This can be understood as a set of scalar multiplications on the vectors and vector additions.

While considering the submatrices as vectors is a more intuitive way to understand our construction, we include a "non-vectorized" explanation here. Since our multiplications and additions are performed in a block-wise fashion, the same number should be multiplied to all the elements in the sub-matrix. E.g., for encoding the first parity node, $a_{1,1}$ should be multiplied with all elements in $\mathbf{A}_1$; $a_{1,2}$ should be multiplied with all elements in $\mathbf{A}_2$, and so on. Since each submatrix $\mathbf{A}_i$ has N/m rows, we have to expand the encoding matrix $G_{\mathbf{A}}$ by N/m as follows:

$$\mathcal{G}_{\mathbf{A}} = G_{\mathbf{A}} \otimes I_{\frac{N}{m} \times \frac{N}{m}}. \tag{3.31}$$

Now, $\mathcal{G}_{\mathbf{A}}$ is a matrix of dimension $(m^2 + p)\frac{N}{m} \times N$, and (3.27) can be rewritten as:

$$\begin{bmatrix} \widetilde{\mathbf{A}}_1 \\ \vdots \\ \widetilde{\mathbf{A}}_{m^2+p} \end{bmatrix} = \mathcal{G}_{\mathbf{A}} \begin{bmatrix} \mathbf{A_1} \\ \vdots \\ \mathbf{A_m} \end{bmatrix}. \tag{3.32}$$

We can construct different codes by choosing different coefficients in $P_{\mathbf{A}}$ and $P_{\mathbf{B}}$. Our code constructions provided in the following will use this general framework and, we will highlight only how $P_{\mathbf{A}}$ and $P_{\mathbf{B}}$ are constructed.

3.1.2.5 Random Code Construction and Probabilistic Guarantees

Construction 3.1.5 (Random Code)**.** *Following the general framework given in* (3.30)*, all entries in $P_{\mathbf{A}}$ and $P_{\mathbf{B}}$ are drawn iid from the standard Gaussian distribution $\mathcal{N}(0, 1)$.*

Theorem 3.1.5. *Construction 3.1.5 provides a systematic MDS matrix-multiplication code with*

probability 1, i.e., the results from any m^2 out of the overall $m^2 + p$ nodes are sufficient to reconstruct the final result $\mathbf{C}$.

To prove the theorem, we need two lemmas.

Lemma 3.1.2 (Corollary 3, p.319 in [77])**.** *A matrix G is an encoding matrix of a systematic MDS code if and only if every square submatrix of the parity generating submatrix G_P is non-singular.*

Lemma 3.1.3. *If the entries of $P_{\mathbf{A}}$ and $P_{\mathbf{B}}$ are drawn iid from the standard Gaussian distribution, every square submatrix of the parity generating submatrix $P_{\mathbf{A}} \star P_{\mathbf{B}}$ is non-singular with probability* 1.

Proof. We will first show that the determinants of any $r \times r$ submatrix ($r \leqslant p$) are non-zero polynomials by mathematical induction. When $r = 1$, this is trivial. Now, assume that every $(r-1) \times (r-1)$ submatrix of $P_{\mathbf{A}} \star P_{\mathbf{B}}$ has a non-zero determinant. Let us denote an arbitrary $r \times r$ submatrix as:

$$S = \begin{bmatrix} a_{i_1,j_1}b_{i_1,k_1} & a_{i_1,j_2}b_{i_1,k_2} & \cdots & a_{i_1,j_r}b_{i_1,k_r} \\ a_{i_2,j_1}b_{i_2,k_1} & a_{i_2,j_2}b_{i_2,k_2} & \cdots & a_{i_2,j_r}b_{i_2,k_r} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_r,j_1}b_{i_r,k_1} & a_{i_r,j_2}b_{i_r,k_2} & \cdots & a_{i_r,j_r}b_{i_r,k_r} \end{bmatrix}. \tag{3.33}$$

The determinant of this matrix can be written as:

$$\det(S) = a_{i_1,j_1}b_{i_1,k_1}D_1 + a_{i_1,j_2}b_{i_1,k_2}D_2 + \cdots + a_{i_1,j_r}b_{i_1,k_r}D_r, \tag{3.34}$$

where D_i is the determinant of the $(r-1) \times (r-1)$ submatrix without the first row and the i-th column of the matrix S, and they are non-zero polynomials due to the induction assumption. Because $(j_1, k_1), (j_2, k_2), \cdots, (j_r, k_r)$ are all distinct, r terms in (3.34) cannot cancel each other out. Hence, $\det(S)$ is not a zero polynomial.

It is easy to see that the set of $a_{i,j}, b_{i,k}$'s in matrix S such that $\det(S) = 0$ is a measure-0 subset of the entire space[1]. For a given r, there are $\binom{m^2}{r} \cdot \binom{p}{r}$ possible submatrices. Let us call

[1]Depending on which rows and columns are chosen for the submatrix S, the entire space can be as small as $\mathbb{R}^{r^2}$ and as big as $\mathbb{R}^{2r^2}$.

the set of $a_{i,j}, b_{i,k}$'s that makes any square submatrix of $P_{\mathbf{A}} \star P_{\mathbf{B}}$ to have determinant 0, a "bad set". The bad set is a union of $\sum_{r=1}^{p} \binom{m^2}{r} \cdot \binom{p}{r}$ measure-0 subsets. Hence, $P(\text{bad set}) = 0$ when $a_{i,j}, b_{i,k}$'s are chosen randomly from a Gaussian distribution. ■

From Lemmas 3.1.2 and 3.1.3, Theorem 3.1.5 follows.

3.1.2.6 Bivariate Polynomial Code Construction

Let us denote a Vandermonde matrix as follows:

$$\text{Vand}_d(u_1, u_2, \cdots u_k) = \begin{bmatrix} 1 & u_1 & \cdots & u_1^{d-1} \\ 1 & u_2 & \cdots & u_2^{d-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & u_k & \cdots & u_k^{d-1} \end{bmatrix}.$$

Construction 3.1.6 (Bivariate Polynomial Code)**.** *Let*

$$\begin{aligned} \mathcal{A} = \text{Vand}_m(\alpha_1, \cdots, \alpha_m),\ \mathcal{B} = \text{Vand}_m(\beta_1, \cdots, \beta_m), \\ \mathcal{A}_P = \text{Vand}_m(\alpha_{m+1}, \cdots, \alpha_{m+p}), \\ \mathcal{B}_P = \text{Vand}_m(\beta_{m+1}, \cdots, \beta_{m+p}), \end{aligned} \tag{3.35}$$

where α_i's and β_i's ($i = 1, \cdots, m^2 + p$) drawn iid from the standard Gaussian distribution.

Following the general framework given in (3.30)*, $P_{\mathbf{A}}$ and $P_{\mathbf{B}}$ are constructed as follows:*

$$P_{\mathbf{A}} = \mathcal{A}_P \mathcal{A}^{-1}, P_{\mathbf{B}} = \mathcal{B}_P \mathcal{B}^{-1}. \tag{3.36}$$

The following lemma explains how Construction 3.1.6 is based on polynomials.

Lemma 3.1.4. *If $\alpha_1, \cdots, \alpha_m$ and $\beta_1, \cdots, \beta_m$ are drawn iid from the standard Gaussian distribution, with probability* 1*, there exists a polynomial of degree $2m - 2$, $\mathbf{h}(x, y)$ that satisfies*

$$\mathbf{h}(\alpha_i, \beta_j) = \mathbf{A}_i \mathbf{B}_j, \tag{3.37}$$

for $i, j = 1, \cdots m$.

Proof. If α_i's are all distinct, we can construct a polynomial $\mathbf{f}(x)$ as follows:

$$\mathbf{f}(x) = \sum_{i=1}^{m} \mathbf{A}_i \prod_{j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j}. \tag{3.38}$$

Similarly, if β_i's are all distinct, we can construct a polynomial $\mathbf{g}(x)$ as follows:

$$\mathbf{g}(x) = \sum_{i=1}^{m} \mathbf{B}_i \prod_{j \neq i} \frac{x - \beta_j}{\beta_i - \beta_j}. \tag{3.39}$$

Then, if we let $\mathbf{h}(x, y) = \mathbf{f}(x)\mathbf{g}(y)$, (3.37) is satisfied. For iid samples from Gaussian distribution, $\Pr(\alpha_i = \alpha_j) = 0$ for $i \neq j$. Hence, the polynomial $\mathbf{h}$ exists with probability 1. ■

Since the degree of the polynomials $\mathbf{f}$ and $\mathbf{g}$ we constructed in (3.38) and (3.39) is $m - 1$, let us write them as follows:

$$\mathbf{f}(x) = f_0 + f_1 x + \cdots + f_{m-1} x^{m-1}, \tag{3.40}$$

$$\mathbf{g}(x) = g_0 + g_1 x + \cdots + g_{m-1} x^{m-1}. \tag{3.41}$$

Because $\mathbf{f}(\alpha_i) = \mathbf{A}_i$ for $i = 1, 2, \ldots, m$, we have:

$$\begin{bmatrix} f_0 \\ \vdots \\ f_{m-1} \end{bmatrix} = \mathcal{A}^{-1} \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_m \end{bmatrix}. \tag{3.42}$$

For the parity nodes, we encode $\mathbf{A}$ and $\mathbf{B}$ using polynomial evaluations $\mathbf{f}(\alpha_i)$ and $\mathbf{g}(\beta_i)$, $i = m + 1, \ldots, m + p$, and let each parity node compute:

$$\mathbf{h}(\alpha_i, \beta_i) = \mathbf{f}(\alpha_i)\mathbf{g}(\beta_i). \tag{3.43}$$

Using this, our encoding matrix can be written as:

$$G_{\mathbf{A}} = \begin{bmatrix} I_{m \times m} \otimes \mathbf{1}_{m \times 1} \\ \hline \mathcal{A}_P \mathcal{A}^{-1} \end{bmatrix}. \tag{3.44}$$

The bottom submatrix $P_{\mathbf{A}} = \mathcal{A}_P \mathcal{A}^{-1}$ is the result of polynomial encoding at the parity nodes given in (3.43). $P_{\mathbf{B}}$ can be obtained similarly.

Theorem 3.1.6. *Construction 3.1.6 provides a systematic MDS matrix-multiplication code with probability 1, i.e., the results from any m^2 out of the overall $m^2 + p$ nodes are sufficient to reconstruct the final result* $\mathbf{C}$.

Proof. First, notice that if we can reconstruct the coefficients f_i's and g_j's in polynomial $h(x, y)$, we can recover $\mathbf{C}$ by evaluating $h(x, y)$ at $x = \alpha_i, y = \beta_j$ for $i, j = 1, \cdots, m$. Hence, we will prove that we can reconstruct the polynomial $h(x, y)$ from any m^2 nodes with probability 1, *i.e.,* any $m^2 \times m^2$ submatrix of the following matrix is invertible:

$$H = \left[\begin{array}{c} \mathcal{A} \otimes \mathcal{B} \\ \hline \mathcal{A}_P \star \mathcal{B}_P \end{array}\right]_{(m^2+p)\times m^2}. \tag{3.45}$$

Denote an arbitrary $m^2 \times m^2$ square submatrix of H by S. We will show that $\det(S)$ is a non-zero polynomial of the standard Gaussian random variables α_i's and β_j's, and hence $\Pr(\det(S) = 0) = 0$. We will use $\mathbf{0}$ to denote a zero polynomial.

Let us rewrite S as:

$$S = \left[\begin{array}{c} S_{sys} \\ \hline S_{par} \end{array}\right],$$

where S_{sys} and S_{par} are from rows of $\mathcal{A}\otimes\mathcal{B}$ and $\mathcal{A}_P \star \mathcal{B}_P$, respectively. Let us denote the number of rows in S_{sys} and S_{par} as σ and ρ.

Case 1: $\sigma = m^2$ and $\rho = 0$. In other words, $S = \mathcal{A} \otimes \mathcal{B}$. Then, from the property of Kronecker product,

$$\begin{aligned} \det(S) &= \det(\mathcal{A})^m \det(\mathcal{B})^m \\ &= \prod_{i\neq j}(\alpha_i - \alpha_j)^m \prod_{i\neq j}(\beta_i - \beta_j)^m, \end{aligned}$$

which is a non-zero polynomial.

Case 2: $1 \leqslant \rho \leqslant p$. We will use induction on ρ.

i) $\rho = 1$.

In this case, S_{par} is a row vector of the following form:

$$S_{par} = \left[\text{Vand}_m(\alpha_k) \otimes \text{Vand}_m(\beta_k) \right],$$

where $k > m$. Using this row vector, the determinant can be expanded as follows:

$$\det(S) = \det(S_1) - \beta_k \det(S_2) + \cdots - \alpha_k^{m-1}\beta_k^{m-1} \det(S_{m^2}), \tag{3.46}$$

where S_i's are submatrices of S excluding the i-th column and the m^2-th row. The signs in (3.46) assume that m is even, but the proof holds the same for an odd m. Notice that $\det(S_1), \cdots, \det(S_{m^2})$ are polynomials only in α_i's and β_j's for $i, j = 1, \cdots m$, and they do not have any α_k or β_k terms for $k > m$. Hence, $\det(S) = \mathbf{0}$ only when

$$\det(S_1) = \cdots = \det(S_{m^2}) = \mathbf{0}. \tag{3.47}$$

i.e., when all these are zero polynomials.

Let us denote $I = \{(i, j) | i, j = 1, \cdots, m\}$ and $I(S) \subseteq I$ as a set of indices of α_i, β_j that are included in S_{sys}. In this case, we have only one element in $I \backslash I(S)$ and let us denote the element as $(\tilde{i}, \tilde{j})$. Now, let us define another matrix S' by replacing S_{par} with

$$\left[\text{Vand}_m(\alpha_{\tilde{i}}) \otimes \text{Vand}_m(\beta_{\tilde{j}}) \right].$$

Notice that the matrix S' now consists of m^2 rows of the systematic part. Therefore, from Case 1, we get:

$$\begin{aligned} \det(S') &= \det(S_1) - \beta_{\tilde{j}} \det(S_2) + \cdots - \alpha_{\tilde{i}}^{m-1}\beta_{\tilde{j}}^{m-1} \det(S_{m^2}) \\ &= \prod_{i \neq j} (\alpha_i - \alpha_j)^m \prod_{i \neq j} (\beta_i - \beta_j)^m \neq \mathbf{0}. \end{aligned}$$

This contradicts (3.47). Thus, $\det(S) \neq 0$.

ii) Let as assume that $\det(S) \neq 0$ for any $\rho \leqslant k$. Then, showing that this holds for $\rho = k + 1$ is similar to what we did for $\rho = 1$.

$$\det(S) = \det(S_1) - \beta_{k+1} \det(S_2) + \cdots - \alpha_{k+1}^{m-1}\beta_{k+1}^{m-1} \det(S_{m^2}). \tag{3.48}$$

Now, let us assume that $\det(S) = 0$. This implies that (3.47) holds. Let us choose $(\tilde{i}, \tilde{j}) \in I \backslash I(S)$ and construct S' by replacing the last row with

$$\left[\text{Vand}_m(\alpha_{\tilde{i}}) \otimes \text{Vand}_m(\beta_{\tilde{j}}) \right].$$

Then S' has k rows from $\mathcal{A}_P \star \mathcal{B}_P$ and $m^2 - k$ rows from $\mathcal{A} \otimes \mathcal{B}$. Thus, by inductive assumption,

$$\begin{aligned} \det(S') &= \det(S_1) - \beta_{\tilde{j}} \det(S_2) + \cdots - \alpha_{\tilde{i}}^{m-1} \beta_{\tilde{j}}^{m-1} \det(S_{m^2}) \\ &\neq \mathbf{0}. \end{aligned}$$

This contradicts (3.47). Thus $\det(S) \neq 0$. ■

3.1.3 Systematic LRC MatDot Codes

In this section, we discuss systematic[2] LRC MatDot code construction that has both the desired properties: being systematic and being locally-recoverable. Although we assume a master-worker system in this paper, LRC matrix multiplication nodes will be also valuable in a fully-distributed system that does not have a master node. Systematic encoding would be particularly useful in this setting because we can obtain the final computation output by only repairing failed systematic nodes. Assuming that failure rate is low, locality will let us repair a failed systematic node by communicating with only a few other nodes instead of communicating with all the other nodes.

We will first give an example of systematic LRC MatDot codes for $m = 4, r = 3$ with $P = 16$ worker nodes.

Example 3.1.4 (Systematic LRC MatDot Codes with $m = 4, r = 3, P = 16$)**.** *We first split*

[2]The definition of systematic codes follows Definition 3.1.1

matrices $\mathbf{A}$ *and* $\mathbf{B}$ *into* 4 *blocks as follows:*

$$\mathbf{A} = \left[\mathbf{A}_1\ \mathbf{A}_1\ \mathbf{A}_3\ \mathbf{A}_4\right], \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \mathbf{B}_3 \\ \mathbf{B}_4 \end{bmatrix} \tag{3.49}$$

where $\mathbf{A}_i$*'s and* $\mathbf{B}_i$*'s are* $N \times N/4$ *and* $N/4 \times N$ *dimensional submatrices, respectively. Let* $\mathcal{A} = \{\alpha_1, \cdots, \alpha_{16}\}$ *be a set of* 16 *distinct real numbers and let* $\mathcal{A}_1 = \{\alpha_1, \cdots, \alpha_4\}, \cdots, \mathcal{A}_4 = \{\alpha_{13}, \cdots, \alpha_{16}\}$ *be disjoint subsets of* $\mathcal{A}$ *that form a partition of* $\mathcal{A}$*. Let* $g(x)$ *be a polynomial of degree* 4 *which satisfies:* $g(\mathcal{A}_i) = \gamma_i$ *for* $i = 1, \cdots, 4$*. Then, we encode the matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *with the following polynomials:*

$$\begin{aligned} p_{\mathbf{A}}(x) = &\big(\mathbf{A}_1 \frac{x-\alpha_1}{\alpha_2-\alpha_1} + \mathbf{A}_2 \frac{x-\alpha_1}{\alpha_2-\alpha_1}\big) f_1(x) \\ &+ \big(\mathbf{A}_3 \frac{x-\alpha_6}{\alpha_5-\alpha_6} + \mathbf{A}_4 \frac{x-\alpha_5}{\alpha_6-\alpha_5}\big) f_2(x) \end{aligned}$$

$$\begin{aligned} p_{\mathbf{B}}(x) = &\big(\mathbf{B}_1 \frac{x-\alpha_1}{\alpha_2-\alpha_1} + \mathbf{B}_2 \frac{x-\alpha_2}{\alpha_1-\alpha_2}\big) f_1(x) \\ &+ \big(\mathbf{B}_3 \frac{x-\alpha_6}{\alpha_5-\alpha_6} + \mathbf{B}_4 \frac{x-\alpha_5}{\alpha_6-\alpha_5}\big) f_2(x) \end{aligned} \quad \textit{where}$$

$f_1(x) = \lambda_{11} + \lambda_{12} g(x)$ *and* $f_2(x) = \lambda_{21} + \lambda_{22} g(x)$*. The coefficients* λ_{ij}*'s are chosen so that* $f_i(\mathcal{A}_j) = \delta_{ij}$ *for* $i, j = 1, 2$*. They can be obtained by solving:*

$$\begin{bmatrix} \lambda_{11} & \lambda_{12} \\ \lambda_{21} & \lambda_{22} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ \gamma_1 & \gamma_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The i*-th worker node receives the encoded matrices,* $p_{\mathbf{A}}(\alpha_i)$ *and* $p_{\mathbf{B}}(\alpha_i)$*, and then computes the*

following product:

$$\begin{aligned} p_{\mathbf{c}}(x) &= p_{\mathbf{A}}(x)p_{\mathbf{B}}(x) \\ &= \big(\mathbf{A}_1\frac{x-\alpha_1}{\alpha_2-\alpha_1} + \mathbf{A}_2\frac{x-\alpha_1}{\alpha_2-\alpha_1}\big)\big(\mathbf{B}_1\frac{x-\alpha_1}{\alpha_2-\alpha_1} \\ &\quad + \mathbf{B}_2\frac{x-\alpha_2}{\alpha_1-\alpha_2}\big)f_1(x)^2 + \big(\ \cdots\ \big)f_1(x)f_2(x) \\ &+ \big(\mathbf{A}_3\frac{x-\alpha_6}{\alpha_5-\alpha_6} + \mathbf{A}_4\frac{x-\alpha_5}{\alpha_6-\alpha_5}\big)\big(\mathbf{B}_3\frac{x-\alpha_6}{\alpha_5-\alpha_6} \\ &\quad + \mathbf{B}_4\frac{x-\alpha_5}{\alpha_6-\alpha_5}\big)f_2(x)^2 \end{aligned}$$

at $x = \alpha_i$. First, note that the following holds:

$$p_{\mathbf{c}}(\alpha_1) = \mathbf{A}_1\mathbf{B}_1, \ \ p_{\mathbf{c}}(\alpha_2) = \mathbf{A}_2\mathbf{B}_2,$$
$$p_{\mathbf{c}}(\alpha_5) = \mathbf{A}_3\mathbf{B}_3, \ \ p_{\mathbf{c}}(\alpha_6) = \mathbf{A}_4\mathbf{B}_4.$$

Hence, this is a systematic code. The degree of $p_{\mathbf{C}}(\cdot)$ is $2 \cdot 4 + 2 = 10$, so with evaluation at any 11 points, we can recover the coefficients on $p_{\mathbf{C}}(x)$. The recovery threshold $K = 11$.

Now, to examine the locality property, let as assume that node 3 is erased. Because f_1 and f_2 are linear combinations of constant and $g(x)$, they are also constant on each subset $\mathcal{A}_i$. For $i = 1, \cdots, 4$, $p_{\mathbf{C}}(\alpha_i)$ becomes a polynomial of degree 2 in α_i as $f_1(\alpha_i), f_2(\alpha_i)$ are constant for $\alpha_1, \cdots, \alpha_4$. Hence, the coefficients of $p_{\mathbf{C}}$ can be recovered from three evaluations, $p_{\mathbf{C}}(\alpha_1), p_{\mathbf{C}}(\alpha_2)$, and $p_{\mathbf{C}}(\alpha_4)$, and thus the lost matrix $p_{\mathbf{C}}(\alpha_3)$ can be recovered. ■

We now give a construction of systematic LRC MatDot codes for general m and r.

Construction 3.1.7 (Systematic LRC MatDot Codes)**.** *The key idea is to replace $g(x)^{i-1}$ in $p_{\mathbf{A}}(x)$ given in (3.11) with $f_i(x)$ which satisfies $f_i(\mathcal{A}_j) = \delta_{ij}$ ($i, j = 1, \cdots, \frac{2m}{r+1}$), and to replace $\sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{\frac{r+1}{2}(i-1)+j}x^{j-1}$ in (3.11) with Lagrange interpolation ($i = 1, \cdots, \frac{2m}{r+1}$). We do similar replacements for $p_{\mathbf{B}}(x)$ in (3.12). Let us explain this in more detail.*

First, we generate f_i's by linearly combining $1, g(x), \cdots, g(x)^{\frac{2m}{r+1}-1}$: $f_i(x) = \sum_{j=1}^{\frac{2m}{r+1}} \lambda_{ij}g(x)^{j-1}$.

Let us denote $g(\mathcal{A}_i) = \gamma_i$. We obtain λ_{ij}'s by solving the following equation:

$$\Lambda \cdot \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \gamma_1 & \gamma_2 & \cdots & \gamma_{\frac{2m}{r+1}} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_1^{\frac{2m}{r+1}-1} & \gamma_2^{\frac{2m}{r+1}-1} & \cdots & \gamma_{\frac{2m}{r+1}}^{\frac{2m}{r+1}-1} \end{bmatrix} = I_{\frac{2m}{r+1}},$$

where $\Lambda = \begin{bmatrix} \lambda_{ij} \end{bmatrix}$ and $I_{\frac{2m}{r+1}}$ is an identity matrix of dimension $\frac{2m}{r+1} \times \frac{2m}{r+1}$. It is easy to see that by this choice of λ_{ij}'s, $f_i(\mathcal{A}_j) = \delta_{ij}$ for $i, j = 1, \cdots, \frac{2m}{r+1}$.

Let $\widetilde{\mathcal{A}}_i$ be a subset of $\mathcal{A}_i$ which has the first half of the elements, that is,

$$\widetilde{\mathcal{A}}_i = \{\alpha_{(r+1)(i-1)+1}, \cdots, \alpha_{(r+1)(i-1)+\frac{r+1}{2}}\},$$

and let $\widetilde{\mathcal{A}}_i(j) = \widetilde{\mathcal{A}}_i \backslash \{\alpha_{(r+1)(i-1)+j}\}$. Now, let us define $\phi_{ij}(x)$ as follows:

$$\phi_{i,j}(x) = \prod_{\alpha \in \widetilde{\mathcal{A}}_i(j)} \frac{x - \alpha}{\alpha_{(r+1)(i-1)+j} - \alpha}.$$

Then, we encode $\mathbf{A}$ and $\mathbf{B}$ using the following polynomials:

$$\begin{aligned} p_{\mathbf{A}}(x) &= \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_j \phi_{1,j}(x) f_1(x) + \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{\frac{r+1}{2}+j} \phi_{2,j}(x) f_2(x) \\ &\quad + \cdots + \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{m-\frac{r+1}{2}+j} \phi_{\frac{2m}{r+1},j}(x) f_{\frac{2m}{r+1}}(x) \\ &= \sum_{i=1}^{\frac{2m}{r+1}} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{\frac{r+1}{2}(i-1)+j} \phi_{i,j}(x) f_i(x), \end{aligned} \tag{3.50}$$

$$p_{\mathbf{B}}(x) = \sum_{i=1}^{\frac{2m}{r+1}} \sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_{\frac{r+1}{2}(i-1)+j} \phi_{i,j}(x) f_i(x). \tag{3.51}$$

The i-th worker receives the evaluation of $p_{\mathbf{A}}(x)$ and $p_{\mathbf{B}}(x)$ at $x = \alpha_i$. A worker node then

computes the following product:

$$\begin{aligned} p_{\mathbf{C}}(x) &= p_{\mathbf{A}}(x)p_{\mathbf{B}}(x) \\ &= \Big(\sum_{i=1}^{\frac{2m}{r+1}}\sum_{j=1}^{\frac{r+1}{2}} \mathbf{A}_{\frac{r+1}{2}(i-1)+j}\phi_{i,j}(x)f_i(x)\Big) \\ &\quad \Big(\sum_{i=1}^{\frac{2m}{r+1}}\sum_{j=1}^{\frac{r+1}{2}} \mathbf{B}_{\frac{r+1}{2}(i-1)+j}\phi_{i,j}(x)f_i(x)\Big), \end{aligned}$$

and returns the result to the master node. ■

The following theorem shows that the Construction 3.1.7 is indeed systematic, and achieves the same locality and recovery threshold as the non-systematic version LRC MatDot codes.

Theorem 3.1.7. *The systematic LRC MatDot code given in Construction 3.1.7 is systematic, and achieves locality r and recovery threshold $K = 4m - r - 2$.*

Proof. To show that the construction is systematic, we have to show that there exists a subset $\mathcal{A}_{\text{sys}} = \{\beta_1, \cdots, \beta_m\} \subseteq \mathcal{A}$ such that $p_{\mathbf{C}}(\beta_i) = \mathbf{A}_i\mathbf{B}_i$. Let $\mathcal{A}_{\text{sys}} = \widetilde{\mathcal{A}}_1 \bigcup \cdots \bigcup \widetilde{\mathcal{A}}_{\frac{2m}{r+1}}$. Now, notice that for $i = 1, \cdots, \frac{2m}{r+1}$ and $j = 1, \cdots, \frac{r+1}{2}$,

$$\begin{aligned} p_{\mathbf{C}}(\alpha_{(r+1)(i-1)+j}) &= p_{\mathbf{A}}(\alpha_{(r+1)(i-1)+j})p_{\mathbf{B}}(\alpha_{(r+1)(i-1)+j}) \\ &= \mathbf{A}_{\frac{r+1}{2}(i-1)+j}\mathbf{B}_{\frac{r+1}{2}(i-1)+j}, \end{aligned}$$

and $\alpha_{(r+1)(i-1)+j} \in \mathcal{A}_{\text{sys}}$. This proves that the code is systematic.

The degree of $\phi_{i,j}$'s is $\frac{r-1}{2}$ and the degree of f_i's is $(r+1)(\frac{2m}{r+1} - 1) = 2m - r - 1$. Thus, the degree of $p_{\mathbf{C}}(x)$ is $2 \cdot (\frac{r-1}{2} + 2m - r - 1) = 4m - r - 3$. This shows that the recovery threshold $K = 4m - r - 2$.

Finally, let us show the locality property. Let $p_{\mathbf{C}}(\alpha)$ be the lost symbol and let $\alpha \in \mathcal{A}_l$. Then, for all $\beta \in \mathcal{A}_l$, $f_i(\beta) = \sum_{j=1}^{\frac{2m}{r+1}} \lambda_{ij}\gamma_l^{j-1} = \psi_i$. Then, $p_{\mathbf{C}}(\beta)$ can be rewritten as:

$$\begin{aligned} p_{\mathbf{C}}(\beta) = &\Big(\sum_{j=1}^{\frac{r+1}{2}} \big(\sum_{i=1}^{\frac{2m}{r+1}} \psi_i\mathbf{A}_{\frac{r+1}{2}(i-1)+j}\big)\phi_{i,j}(\beta)\Big)\cdot \\ &\Big(\sum_{j=1}^{\frac{r+1}{2}} \big(\sum_{i=1}^{\frac{2m}{r+1}} \psi_i\mathbf{B}_{\frac{r+1}{2}(i-1)+j}\big)\phi_{i,j}(\beta)\Big). \end{aligned}$$

Since $\phi_{i,j}$'s are polynomials of degree $\frac{r-1}{2}$, $p_{\mathbf{C}}(\beta)$ is a degree-$(r-1)$ polynomial in β. Hence, from the evaluation of $p_{\mathbf{C}}(\cdot)$ at the r points in $\mathcal{A}_l \backslash \{\alpha\}$, we can recover the coefficients of $p_{\mathbf{C}}(\cdot)$. The lost symbol $p_{\mathbf{C}}(\alpha)$ can then be recovered by evaluating $p_{\mathbf{C}}(\beta)$ at $\beta = \alpha$. ■

3.2 Masterless Coded Computing Strategies

3.2.1 Coded SUMMA

3.2.1.1 System Model and Notations

The computation goal is to compute a matrix product:

$$\mathbf{C} = \mathbf{A}\mathbf{B}, \tag{3.52}$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{m \times m}$. The matrices need not be the same dimensions in our algorithms, but this is just for simplicity. We have p nodes under the masterless setting. We call systematic nodes for the nodes that have the original data and parity nodes for those that have encoded data. We also assume that the code rate is fixed, which is defined as:

$$\frac{\text{(\# systematic nodes)}}{\text{(\# systematic nodes)} + \text{(\# parity nodes)}}$$

Depending on the algorithm, p nodes will be placed on a 2D or 3D grid, and we will use $P(i,j)$ ($P(i,j,l)$) to denote the (i,j)-th ((i,j,l)-th) node on the grid. We consider hard failures in this paper, and we assume that we cannot recover any data from a failed node.

For communication cost analysis, we assume the α-β model. For an $m \times m$ matrix $\mathbf{X}$, we will use $\mathbf{X}_i^{\text{row}}$ ($\mathbf{X}_i^{\text{col}}$) to denote the i-th row block (column block) of $\mathbf{X}$, and $\mathbf{X}_{i,j}$ to denote the (i,j)-th sub-block of $\mathbf{X}$.Finally, we define *recovery threshold* as the minimum number of workers required to reconstruct $\mathbf{C}$ in the worst case.

Algorithm 1 SUMMA

Initial Data Distribution: $P(i,j)$ has $\mathbf{A}_{i,j}$ and $\mathbf{B}_{i,j}$.

for $k = 1$ **to** $\sqrt{p}$ **do**

/* For $i = 1, \ldots, \sqrt{p}$ in parallel */

$P(i,k)$ broadcasts $\mathbf{A}_{i,k}$ to the i-th row

/* For $j = 1, \ldots, \sqrt{p}$ in parallel */

$P(k,j)$ broadcasts $\mathbf{B}_{k,j}$ to the j-th column

/* For all nodes in parallel */

$P(i,j)$ computes $\mathbf{C}_{i,j} \leftarrow \mathbf{C}_{i,j} + \mathbf{A}_{i,k}\mathbf{B}_{k,j}$

end for

3.2.1.2 Background: SUMMA and 2.5D SUMMA

The scalable universal matrix multiplication algorithm (SUMMA) is a parallel algorithm for general matrix multiplication, which is very simple and highly efficient [113]. The SUMMA assumes a 2D grid placement of nodes and computes $\mathbf{C} = \mathbf{AB}$ through a series of outer product updates.

Let $\mathbf{A} = \begin{bmatrix} \mathbf{A}_1^{\text{col}} & \cdots & \mathbf{A}_{\sqrt{p}}^{\text{col}} \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} \mathbf{B}_1^{\text{row}} \\ \vdots \\ \mathbf{B}_{\sqrt{p}}^{\text{row}} \end{bmatrix}$. Then, $\mathbf{C} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_k\mathbf{B}_k$. In the k-th iteration of SUMMA, we compute one outer product, $\mathbf{A}_k\mathbf{B}_k$. More details are given in Algorithm 1.

To further increase parallelism, 2.5D SUMMA was proposed [103], which adds one more dimension to the grid. 2.5D SUMMA arranges p nodes into a $\sqrt{p/c} \times \sqrt{p/c} \times c$ grid, and each $\sqrt{p/c} \times \sqrt{p/c}$ layer computes a c-th fraction of SUMMA. Then, the results of all the layers are aggregated to compute: $\mathbf{C} = \sum_{k=1}^{c} \mathbf{A}_k^{\text{col}}\mathbf{B}_k^{\text{row}}$. The full algorithm is summarized in Algorithm 2.

3.2.1.3 Problem Definition

Given the system model given in Section 3.2.1.1, we want to construct a coding strategy for SUMMA and 2.5D SUMMA (Algorithm 1,2) that has small encoding and decoding communication overhead.

3.2.1.4 Coded SUMMA

For algorithms like SUMMA which place nodes on a 2D grid, a natural choice of codes is ABFT/Product codes [52, 67]. In this section, we provide insights on how ABFT/Product codes can be incorporated into SUMMA's data distribution and communication patterns. We will call coded SUMMA algorithm with an $n \times n$ grid with a $k \times k$ grid of systematic nodes as (n, k) coded SUMMA.

Example 3.2.1 ($(3, 2)$ Coded SUMMA)**.** *Let us consider using the top-right 2x2 grid as systematic nodes, and the remaining nodes as parity nodes. Matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *are divided into 3x3 sub-blocks, and node* $P(i, j)$ *initially has* $\mathbf{A}_{i,j}$ *and* $\mathbf{B}_{i,j}$*.*

Encoding: *For encoding* $\mathbf{A}$*, let us further break down* $\mathbf{A}_{i,j}$ *into two equal-sized row blocks, and denote them* $\mathbf{A}^{(1)}_{i,j}$ *and* $\mathbf{A}^{(2)}_{i,j}$*. Then, each node encodes parity symbols by computing* $\mathbf{A}^{(3)}_{i,j} = \mathbf{A}^{(1)}_{i,j} + \mathbf{A}^{(2)}_{i,j}$*, and each column performs column-wise all-to-all communication to shuffle the row blocks of* $\mathbf{A}$*. After shuffling, node* $P(i, j)$ *will have* $\widetilde{\mathbf{A}}_{i,j}$ *that is defined as:*

$$\widetilde{\mathbf{A}}_{i,j} = \begin{bmatrix} \mathbf{A}^{(i)}_{1,j} \\ \mathbf{A}^{(i)}_{2,j} \\ \mathbf{A}^{(i)}_{3,j} \end{bmatrix} \tag{3.53}$$

For instance, node $P(1, 1)$ *and* $P(3, 1)$ *will have:*

$$\widetilde{\mathbf{A}}_{1,1} = \begin{bmatrix} \mathbf{A}^{(1)}_{1,1} \\ \mathbf{A}^{(1)}_{2,1} \\ \mathbf{A}^{(1)}_{3,1} \end{bmatrix}, \widetilde{\mathbf{A}}_{3,1} = \begin{bmatrix} \mathbf{A}^{(3)}_{1,1} \\ \mathbf{A}^{(3)}_{2,1} \\ \mathbf{A}^{(3)}_{3,1} \end{bmatrix},$$

respectively.

Encoding $\mathbf{B}$ *is similar, but performed in a column-wise fashion.* $\mathbf{B}_{i,j}$ *is split into two equal-sized column blocks, i.e.,* $\mathbf{B}_{i,j} = \begin{bmatrix} \mathbf{B}_{i,j}^{(1)} & \mathbf{B}_{i,j}^{(2)} \end{bmatrix}$*. Then, each node computes:* $\mathbf{B}_{i,j}^{(3)} = \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)}$*. Each row then performs row-wise all-to-all communication to shuffle the column blocks of* $\mathbf{B}$ *so that after shuffling, node* $P(i,j)$ *has:*

$$\widetilde{\mathbf{B}}_{i,j} = \begin{bmatrix} \mathbf{B}_{i,1}^{(j)} & \mathbf{B}_{i,2}^{(j)} & \mathbf{B}_{i,3}^{(j)} \end{bmatrix}. \tag{3.54}$$

Computation: *We perform SUMMA given in Algorithm 1 on the rearranged matrix* $\widetilde{\mathbf{A}}$ *and* $\widetilde{\mathbf{B}}$*.*

Decoding: *The decoding algorithm is detailed in Algorithm 3. Notice that the shuffled rows and columns are automatically rearranged back to* $\mathbf{AB}$ *during the decoding process. We can further optimize the decoding process by doing "lazy decoding", which is explained later in the section.* ■

The SUMMA can be thought of as running a series of generic matrix multiplications where we compute:

$$\mathbf{C} \leftarrow \alpha\mathbf{C} + \beta\mathbf{AB}, \tag{3.55}$$

for $\sqrt{p}$ iterations. The proposed coding technique can thus be applied to any iterative matrix-multiply updates.

Remark 3.2.1. [Lazy Decoding] For iterative algorithms of k iterations, there exist two ways of decoding: decode after each iteration or decode after all k iterations. We propose *lazy decoding* which is in between the two extremes.

The goal of lazy decoding is to decode in the middle of iterative matrix-multiplies only when the decoding is necessary. We will consider that decoding is necessary if any failure in the next iteration can make the results undecodable. The condition for decodability of product codes was given in [60, 67]. If we represent failure patterns as a bipartite graph where left and right vertices correspond to rows and columns on the grid, respectively. There is an edge between the i-th left vertex and the j-th right vertex, if the node $P(i,j)$ is a failure. Then, the code is undecodable if

there exists a subgraph where each node has degree greater than $n-k$. Now, we will show how the lazy decoding approach can be implemented in a fully-distributed manner.

After computing one iteration of (3.55), all the successful nodes will broadcast their status so that all the surviving nodes can construct the bipartite graph of the failure pattern of the entire grid.

Let $P(i,j)$ be one of the surviving nodes. Then, $P(i,j)$ adds an edge between the i-th left vertex and the j-th right vertex on the graph, and check the decodability condition. If the pattern is undecodable, it will broadcast to all the nodes on the grid that initiates decoding process. After recovering all the nodes on the $n \times n$ grid, the next iteration resumes. We will not specify implementation details of this, but note that each only has to send one-bit beacon, which is much cheaper than exchanging matrices.

3.2.1.5 Communication and Computation Cost Analysis

We analyze the communication cost and computation cost in coded SUMMA. The coded SUMMA has three stages, *i.e.,* encoding, computation, and decoding. Denote the time of these three parts respectively by $T^{\text{enc}}_{\text{SUMMA}}$, $T^{\text{comp}}_{\text{SUMMA}}$, and $T^{\text{dec}}_{\text{SUMMA}}$.

Theorem 3.2.1. *For the system model given in Section 3.2.1.1, communication time of the* (n,k) *coded SUMMA is given as follows:*

$$T^{enc}_{SUMMA} = \alpha\Theta(\log n) + \beta\Theta(m^2 \log n/n^2), \tag{3.56}$$

$$T^{comp}_{SUMMA} = \alpha\Theta(n \log n) + \beta\Theta(m^2/n), \tag{3.57}$$

$$T^{dec}_{SUMMA} = \alpha\Theta(\log n) + \beta\Theta(m^2 \log n/n^2). \tag{3.58}$$

Thus, the encoding and decoding time in coded SUMMA is negligible if $n \gg 1$.

Proof. First, we look at the encoding time. The encoding on each one of $\mathbf{A}$ and $\mathbf{B}$ can be conducted by a local encoding step followed by a shuffling step. At the beginning, each node has

data of size m^2/n^2. The data is partitioned into k small blocks and encoded into n small blocks. Thus, the data at each node after encoding is m^2/nk. Then, the data are shuffled in the n nodes using an all-to-all communication. We use the communication efficient all-to-all algorithm in [11] on the row direction for $\mathbf{B}$ and the column direction for $\mathbf{A}$. If each node has u bits and the number of nodes is v, this algorithm completes in $2\log v$ communication rounds, and requires sending $u\log v$ bits in total. Note that we have a factor 2 because we need to encode both $\mathbf{A}$ and $\mathbf{B}$.

Local encoding: $C_{\text{SUMMA}}^{\text{encoding}} = 2m^2/n^2/k \cdot (nk) = 2m^2/n$.

Shuffling (all-to-all): $T_{\text{SUMMA}}^{\text{encoding}} = 2\alpha\log n + 4\beta(m^2/nk)\log n = \alpha\Theta(\log n) + \beta\Theta(m^2\log n/n^2)$.

Second, we look at the SUMMA computing time. The computation proceeds in n iterations. In each iteration, one node broadcasts data of size m^2/nk to a row, and one node broadcasts data of the same size to a column. Then, local matrix-multiplication is conducted.

Broadcast: $T_{\text{SUMMA}}^{\text{computing}} = [4\alpha\log n + 4\beta(m^2/nk)] \cdot n = \alpha\Theta(n\log n) + \beta\Theta(m^2/n)$.

Local computing: $C_{\text{SUMMA}}^{\text{computing}} = n \cdot (m/k)^2 \cdot (m/n) = m^3/k^2$.

Finally, we look at the SUMMA decoding time. All the rows in the 2D mesh perform decoding in parallel. Then, all the columns perform decoding in parallel. One round of decoding requires two all-to-all communication steps. The data at each node (partial matrix of C) has size m^2/k^2.

Shuffling (all-to-all): $T_{\text{SUMMA}}^{\text{decoding}} = 2\alpha\log n + 4\beta(m^2/k^2)\log n = \alpha\Theta(\log n) + \beta\Theta(m^2\log n/n^2)$.

■

3.2.1.6 Coded 2.5D SUMMA

In 2.5D SUMMA, each layer computes a different set of outer products, that is, the l-th layer computes $\mathbf{A}_l^{\text{col}}\mathbf{B}_l^{\text{col}}$ where $\mathbf{A} = \begin{bmatrix}\mathbf{A}_1^{\text{col}} & \cdots & \mathbf{A}_c^{\text{col}}\end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix}\mathbf{B}_1^{\text{row}} \\ \vdots \\ \mathbf{B}_c^{\text{row}}\end{bmatrix}$, and the final product is the sum of all the outer products computed at different layers: $\mathbf{C} = \sum_{k=1}^{c}\mathbf{A}_l^{\text{col}}\mathbf{B}_l^{\text{col}}$. For this

last dimension where we split the outer products, we can apply MatDot codes [29] since MatDot codes split the matrix product into outer products. Hence, we propose coded 2.5D SUMMA which applies ABFT/Product codes within each layer and MatDot codes across the layers. We will use $(n, k) \times (N, K)$ coded 2.5D SUMMA to denote the coded strategy which uses (n, k) coded SUMMA at each layer and has K systematic layers out of total of N layers.

A Concise description of the main algorithm. (Fig. 3.4) We will call the three directions in the 3D grid direction-i, -j, and -l respectively.

Step 1 (Encoding across layers) $\mathbf{A}$ and $\mathbf{B}$ are encoded locally using MatDot codes on the first processor layer. Then, the encoded matrices are scattered in direction-l so that each layer gets the same linear combination.

Step 2 (Encoding on rows and columns) For all layers in parallel, encode on partial $\mathbf{A}$ and $\mathbf{B}$ using product codes. Then, encoded submatrices are shuffled to ensure that each node has the same single linear combinatinon. The MDS codes and MatDot codes are encoded on orthogonal directions.

Step 3 (SUMMA) For all layers in parallel, the encoded data is gathered into the minimum amount of nodes ($\frac{1}{N}$-fraction of all the nodes). Matrix-matrix multiplications are conducted using the SUMMA algorithm on all layers. The gathering step is for reducing the number of stages in SUMMA.

Step 4 (Decoding) In the case of failures, decode on the rows, columns, or across layers.

Let us now provide a simple example of $(3, 2) \times (4, 2)$ coded 2.5D SUMMA.

Example 3.2.2 ($(3, 2) \times (4, 2)$ coded 2.5D SUMMA)**.** *The node $P(i, j, 1)$ initially has $\mathbf{A}_{i,j}$ and $\mathbf{B}_{i,j}$.*

Encoding: *We will first encode MatDot codes and begin with splitting $\mathbf{A}_{i,j}$ into two smaller*

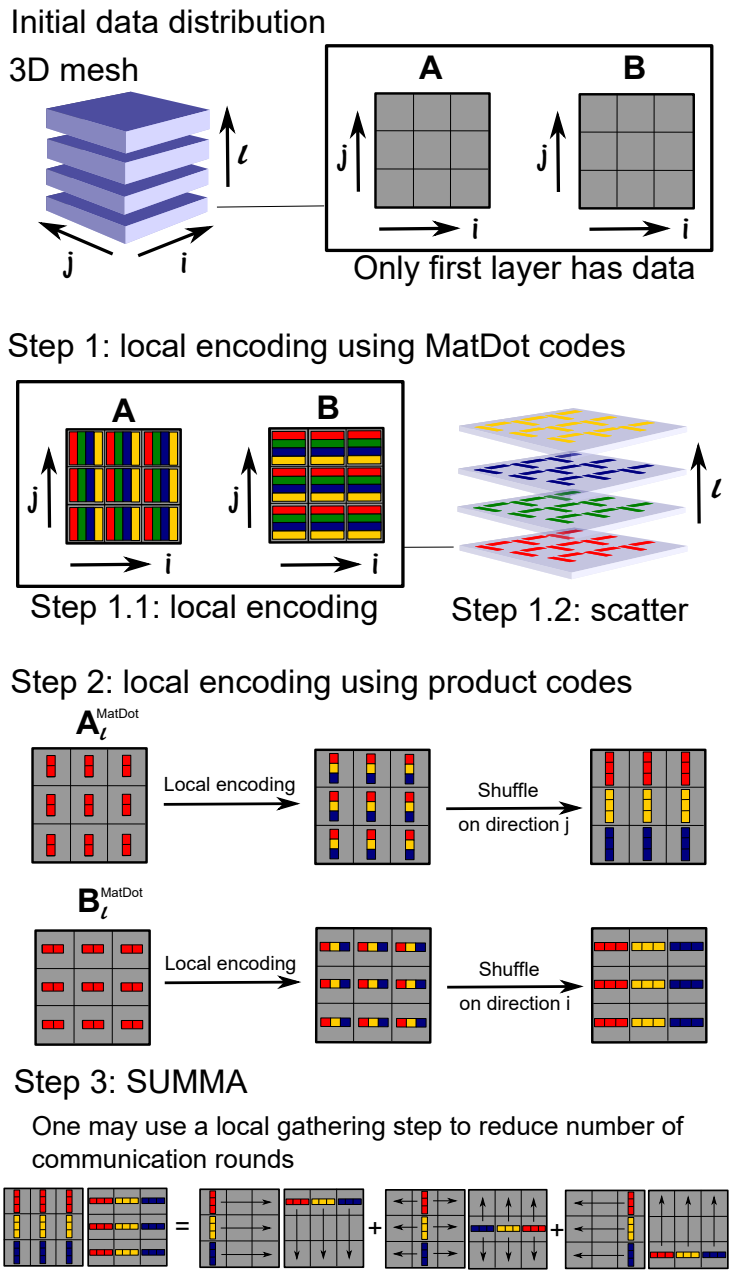


Figure 3.4: Coded 2.5D SUMMA algorithm

column blocks and $\mathbf{B}_{i,j}$ *into two smaller row blocks as follows:*

$$\mathbf{A}_{i,j} = \begin{bmatrix} \mathbf{A}_{i,j}^{(1)} & \mathbf{A}_{i,j}^{(2)} \end{bmatrix}, \mathbf{B}_{i,j} = \begin{bmatrix} \mathbf{B}_{i,j}^{(1)} \\ \mathbf{B}_{i,j}^{(2)} \end{bmatrix}. \tag{3.59}$$

Then, the node $P(i,j,1)$ *locally computes four encoded column-blocks and row-blocks as fol-*

lows:

$$\mathbf{A}_{i,j,1} = \mathbf{A}_{i,j}^{(1)} + \alpha_1 \mathbf{A}_{i,j}^{(2)}, \;\; \mathbf{B}_{i,j,1} = \alpha_1 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)},$$
$$\mathbf{A}_{i,j,2} = \mathbf{A}_{i,j}^{(1)} + \alpha_2 \mathbf{A}_{i,j}^{(2)}, \;\; \mathbf{B}_{i,j,2} = \alpha_2 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)},$$
$$\mathbf{A}_{i,j,3} = \mathbf{A}_{i,j}^{(1)} + \alpha_3 \mathbf{A}_{i,j}^{(2)}, \;\; \mathbf{B}_{i,j,3} = \alpha_3 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)},$$
$$\mathbf{A}_{i,j,4} = \mathbf{A}_{i,j}^{(1)} + \alpha_4 \mathbf{A}_{i,j}^{(2)}, \;\; \mathbf{B}_{i,j,4} = \alpha_4 \mathbf{B}_{i,j}^{(1)} + \mathbf{B}_{i,j}^{(2)},$$

where $\alpha_1, \cdots, \alpha_4$ are four distinct real numbers. Then $P(i,j,1)$ sends $\mathbf{A}_{i,j,k}$ to $P(i,j,k)$ for $k = 2,3,4$. We can also use systematic MatDot codes where $\mathbf{A}_{i,j,1} = \mathbf{A}_{i,j}^{(1)}$ and $\mathbf{A}_{i,j,1} = \mathbf{A}_{i,j}^{(2)}$.

After MatDot encoding step, the node $P(i,j,k)$ will have $\mathbf{A}_{i,j,k}$ and $\mathbf{B}_{i,j,k}$ for all $i,j = 1,\ldots,3, k = 1,\ldots,4$. Then, each layer will perform encoding for (3,2) coded SUMMA as described in Example 3.2.1.

Computation: *Perform 2.5D SUMMA on the $3 \times 3 \times 4$ grid as given in Algorithm 2.*

Decoding: *We decode MatDot codes across layers. If some node's data is undecodable through MatDot codes, we decode the Product code within the layer.* ■

In the $(n,k) \times (N,K)$ coded 2.5D SUMMA, we encode MatDot codes on shuffled columns of $\mathbf{A}$ and shuffled rows of $\mathbf{B}$. We will describe more on *shuffled MatDot codes* and show that the shuffling does not change the structure of MatDot codes.

Shuffled MatDot codes in $(n,k) \times (N,K)$ coded 2.5D SUMMA. At the first layer, the i-th column generates N encoded column blocks of $\mathbf{A}$ as follows:

$$\sum_{l=1}^{K} \mathbf{A}_i^{(l)} \alpha_\rho^{l-1} \tag{3.60}$$

for $\rho = 1, \ldots N$. Similarly, the i-th row in the first layer encodes $\mathbf{B}$ as follows:

$$\sum_{l=1}^{K} \mathbf{B}_i^{(l)} \alpha_\rho^{K-l} \tag{3.61}$$

for $\rho = 1, \ldots, N$. Note that we use $\mathbf{A}_i$ and $\mathbf{B}_i$ to denote $\mathbf{A}_i^{\text{col}}$ and $\mathbf{B}_i^{\text{row}}$. Now the k-th layer gets encoded blocks of $\mathbf{A}$ and $\mathbf{B}$ as follows:

$$\Big\{ \sum_{l=1}^{K} \mathbf{A}_i^{(l)} \alpha_k^{l-1} \quad \text{for } i = 1, \ldots, K \Big\},$$
$$\Big\{ \sum_{l=1}^{K} \mathbf{B}_i^{(l)} \alpha_k^{K-l} \quad \text{for } i = 1, \ldots, K \Big\}.$$

Then, the k-th layer computes:

$$\sum_{i=1}^{K} \left(\sum_{l=1}^{K} \mathbf{A}_i^{(l)} \alpha_k^{l-1} \right) \left(\sum_{j=1}^{K} \mathbf{B}_i^{(j)} \alpha_k^{K-j} \right). \tag{3.62}$$

Lemma 3.2.1. *Shuffled MatDot codes in $(n,k) \times (N,K)$ 2.5D coded SUMMA can recover any failed node $P(i,j,k)$ from the set of surviving nodes $\mathcal{S} = \{P(i,j,\kappa), \kappa \in [1,\ldots,N]\}$ if $|\mathcal{S}| \geqslant 2K-1$.*

Proof. Let us define a polynomial $f_{\mathbf{C}}(x)$ as:

$$f_{\mathbf{C}}(x) = \sum_{i=1}^{K} \left(\sum_{l=1}^{K} \mathbf{A}_i^{(l)} x^{l-1} \right) \left(\sum_{j=1}^{K} \mathbf{B}_i^{(j)} x^{K-j} \right). \tag{3.63}$$

The coefficient of x^{K-1} in $f_{\mathbf{C}}(x)$ is:

$$\sum_{i=1}^{K} \sum_{l=1}^{K} \mathbf{A}_i^{(l)} \mathbf{B}_i^{(l)} = \mathbf{AB} = \mathbf{C}. \tag{3.64}$$

Since the degree of the polynomial $f_{\mathbf{C}}$ is $2K-2$, with any $2K-1$ evaluations of the polynomial, we can recover all the coefficients including (3.64). ■

Theorem 3.2.2. *The recovery threshold of $(n,k) \times (N,K)$ 2.5D SUMMA is given by:*

$$n^2 N - (N-2K)(n-k+1)^2 + 1. \tag{3.65}$$

Proof. We want to show that the minimum number of failures that cannot be decoded is $\lambda_{min} = (N-2K)(n-k+1)^2$. Let us first show that the worst-case scenario on each layer without MatDot codes is $\psi_{min} = (n-k+1)^2$. Let ψ be the number of failures on $n \times n$ grid. We have

to show that i) any $\psi < \psi_{min}$ is decodable and ii) there exists a pattern of ψ_{min} failures that is not decodable. To show i), let us assume that $\psi \leqslant (n-k+1)^2 - 1$. Let ψ_i be the number of failures at the i-th column and $\mathcal{C}_f = \{i : \psi_i > n-k\}$. Then, $|\mathcal{C}_f| < n-k+1$. Thus, there are at least k columns that are decodable, and hence we can decode the entire grid. To prove ii), consider a scenario where the bottom right $(n-k+1) \times (n-k+1)$ sub-grid fails. Then, we cannot decode the result at (k,k)-th node.

We can use a similar argument to extend this to the $(n,k) \times (N,K)$ 2.5D SUMMA. Let us assume that λ be the total number of failures on the 3D grid, and λ_i be the number of failures at the i-th layer. We have to show that i) any $\lambda < \lambda_{min}$ is decodable and ii) there exists a pattern of λ_{min} failures that is not decodable. Let us assume that $\lambda < \lambda_{min}$. Then $|\mathcal{L}_f = \{i : \lambda_i \geqslant \psi_{min}\}| < N - 2K$. This shows that there are at least $2K+1$ layers with less than ψ_{min} failures, and hence these layers are decodable. From Lemma 3.2.1, we can see that as long as there are $2K+1$ successful layers, we can decode the final output. Now, assume that the first $(N-2K)$ layers have failures at their bottom right $(n-k+1) \times (n-k+1)$ sub-grids. Then, we cannot decode the (k,k)-th node on these layers, and we only have $2K$ layers that have successful (k,k)-th node. Thus, this is not decodable. ■

Remark 3.2.2. The threshold in Theorem 3.2.2 is the minimum number of successful nodes in the worst-case scenario to ensure recovery. There exist scenarios in which the number of successful nodes is smaller than the recovery threshold, but the recovery can still be successful.

3.2.1.7 Communication and Computation Cost Analysis

We analyze the communication cost and computation cost in the coded 2.5D SUMMA algorithm. Again, for simplicity, assume the matrices $\mathbf{A}$ and $\mathbf{B}$ both have size $m \times m$. The processor mesh has size $n \times n \times N$.

Denote by T_{comm} the time required for 2.5D coded SUMMA. Denote by $T_{\text{comm}}^{\text{ABFT}}$ the extra time for the coding cost in product codes. Denote by $T_{\text{comm}}^{\text{MatDot}}$ the extra time for coding cost in MatDot

codes.

Theorem 3.2.3. *Suppose the product codes and MatDot codes have constant rate, i.e.,* $n = \Theta(k)$ *and* $N = \Theta(K)$*. Then,*

$$T_{comm} = \left[\alpha\Theta\left(\log n\right) + \beta\Theta\left(m^2/n^2\right)\right] \cdot \frac{n}{N}, \tag{3.66}$$

$$T_{comm}^{ABFT} = \alpha\Theta(\log n) + \beta\Theta(m^2 \log n/n^2), \tag{3.67}$$

$$T_{comm}^{MatDot} = \alpha\Theta(\log n) + \beta\Theta(m^2/n^2). \tag{3.68}$$

The results lead to the following observations:

- *For product codes:*
  - *(Latency) The latency of encoding and decoding product codes is negligible if* $N = o(n)$.
  - *(Bandwidth) The bandwidth of encoding and decoding product codes is negligible if* $N = o(n/\log n)$*. Note that the* $\log n$ *factor in all-to-all communications can be removed if one uses the ring algorithm. However, the number of communication rounds increase from* $\log n$ *to* n*.*
- *For MatDot codes:*
  - *(Latency) The latency of encoding and decoding MatDot codes is negligible if* $N = o(n)$.
  - *(Bandwidth) The bandwidth of encoding and decoding MatDot codes is negligible if* $N = o(n)$.

Regarding the condition $N = o(n)$, note that the motivation for 2.5D SUMMA instead of 3D SUMMA is that the replication factor cannot be as large as $p^{1/3}$. Thus, in the usual case, we have $N = o(n)$ (otherwise we can use 3D SUMMA in which the data is replicated n times).

Proof. We analyze the time complexity of both communication and computation in each step. W.L.O.G, we only calculate the complexity of multiplications and ignore additions in matrix-matrix multiplications.

Encoding MatDot codes and scattering the coded results

The first layer has $n \times n$ nodes. Each node has a square matrix of size m^2/n^2. Each local square matrix is partitioned into K small blocks and encoded into N small blocks. The N small blocks at a particular node on the first layer is scattered to N layers. Both $\mathbf{A}$ and $\mathbf{B}$ need encoding and scattering.

Local encoding cost: $C_1 = 2m^2/n^2 \cdot N$.

Communication cost (scatter using recursive-halving [110]): $T_1 = 2\alpha \log N + 2\beta \frac{m^2}{n^2} \cdot \frac{N}{K}$.

Encoding product codes and shuffling the encoded data Each node now has two small blocks of size $m^2/n^2/K = \frac{m^2}{n^2 K}$. It further divides each small block into k and encode into n. Thus, the data size at each node becomes $\frac{m^2}{n^2 K} \cdot n/k = \frac{2m^2}{nkK}$. The shuffling stage can use the communication efficient all-to-all algorithm [11] on the row direction for $\mathbf{B}$ and the column direction for $\mathbf{A}$. If each node has u bits and the number of nodes is v, this algorithm completes in $2 \log v$ communication rounds, and requires sending $u \log v$ bits in total.

Local encoding cost: $C_2 = \frac{2m^2}{n^2 K} \cdot \frac{1}{k} \cdot nk = \frac{2m^2}{nK}$.

Communication cost (all-to-all): $T_2 = 4\alpha \log n + \beta \frac{2m^2}{nkK} \log n$

Compute matrix-matrix multiplications using SUMMA The data on each layer is gathered into n^2/K nodes, i.e., the nodes in each row and column are partitioned into groups of size K and a local data gathering is carried out. Then, SUMMA proceeds in n/K rounds. In each round, one node in each row broadcasts data of size $\frac{m^2}{nkK} \cdot K = \frac{m^2}{nk}$ to the entire row, and similarly for each column. Then, local computation is carried out, which multiplies two matrices of size $m/n \times m/k$.

Local gathering using recursive-doubling [110]: $T_{3,\text{gather}} = 2\alpha \log K + \frac{2m^2}{nk}\beta$.

Broadcast in SUMMA (scatter using recursive-halving followed by all-gather using recursive-doubling): $T_{3,\text{bcast}} = (4\alpha \log n + \frac{4m^2}{nk}\beta) \cdot (n/K)$.

Local matrix-matrix multiplication: $C_3 = (m/n \times (m/k)^2) \cdot (n/K) = \frac{m^3}{k^2 K}$.

Decoding and reduction The decoding of product codes requires an all-to-all communication on each row or column to reversely shuffle the computation results. The local partial result at each node has size $m/k \times m/k = m^2/k^2$. The number of nodes that need to participate in the decoding is k. The decoding of MatDot codes only requires a reduce across layers. The data size at each node in the reduction phase is still m^2/k^2, and the number of layers required in the reduce is $2K-1$ (for MatDot codes).

Decoding product codes (all-to-all): $T_{\text{4,a2a}} = 2\alpha \log k + \beta(m^2/k^2)\log k$.

Decoding MatDot codes (reduce using recursive-halving followed by tree-gather [110]): $T_{\text{4,reduce}} = 2\alpha \log(2K-1) + (2m^2/k^2)\beta$.

Note that this communication cost analysis is the worst-case analysis because if we use *systematic* codes, we don't have to communicate at all when there is no failure. Also, we may only need to communicate for the undecodable sysematic nodes on each layer.

Puting all the things together **Overall communication cost:**

The overall communication cost is shown in the following.

$$\begin{aligned}
T_{\text{comm}} =& T_1 + T_2 + T_{\text{3,gather}} + T_{\text{3,bcast}} + T_{\text{4,a2a}} + T_{\text{4,reduce}} \\
=& 2\alpha \log N + 2\beta \frac{m^2}{n^2} \cdot \frac{N}{K} \\
&+ 4\alpha \log n + \beta \frac{2m^2}{nkK} \log n \\
&+ 2\alpha \log K + \frac{2m^2}{nk}\beta \\
&+ (4\alpha \log n + \frac{4m^2}{nk}\beta) \cdot (n/K) \\
&+ 2\alpha \log k + \beta(m^2/k^2)\log k \\
&+ 2\alpha \log(2K-1) + (2m^2/k^2)\beta \\
\overset{(a)}{=}& \left[\alpha\Theta\left(\log n\right) + \beta\Theta\left(m^2/n^2\right)\right] \cdot \frac{n}{N},
\end{aligned}$$

where in step (a), we use the fact that $K = \Theta(N)$ and $k = \Theta(n)$, i.e., the code has constant rate.

Now, we look at the communication time for encoding and decoding only. For product codes,

the extra communication is due to the shuffling in step 2 (see Section 3.2.1.7) and the decoding in step 4 (see Section 3.2.1.7). Thus, the extra communication time due to the use of product codes is

$$\begin{aligned} T_{\text{comm}}^{\text{product-code}} =& T_2 + T_{4,\text{a2a}} \\ =& 4\alpha \log n + \beta \frac{2m^2}{nkK} \log n \\ &+ 2\alpha \log k + \beta (m^2/k^2) \log k \\ =& \alpha \Theta(\log n) + \beta \Theta(m^2/n^2 \cdot \log n). \end{aligned}$$

For MatDot codes, the extra communication comes from the local gathering step (see Section 3.2.1.7) and the decoding step (see Section 3.2.1.7). Thus, the overall communication due to the use of MatDot codes is

$$\begin{aligned} T_{\text{comm}}^{\text{MatDot}} =& T_{3,\text{gather}} + T_{4,\text{reduce}} \\ =& 2\alpha \log K + \frac{2m^2}{nk} \beta \\ &+ 2\alpha \log(2K-1) + (2m^2/k^2)\beta \\ =& \alpha \Theta(\log N) + \beta \Theta(m^2/n^2). \end{aligned}$$

■

In Chapter 4, we will present experimental results for Coded 2.5D SUMMA.

3.2.2 Coded FFT

3.2.2.1 System Model

We assume that we have a total of P processors under the masterless setup. Among P processors, K of them are *"systematic processors"*, and the remaining $P - K$ processors are *"parity processors"* We assume a massively parallel setup where K is very big, but K does not grow faster than $\Theta(\log N / \log \log N)$. For communication latency, we assume the α-β model.

Using P processors, we want to compute N-point FFT:

$$\boldsymbol{Z} = F_N \boldsymbol{x} \tag{3.71}$$

where $\boldsymbol{x}$ is a length-N input data vector, F_N is an N-by-N DFT matrix (ω_N: the N-th root of unity) represented as

$$F_N = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N-1} & \cdots & \omega_N^{(N-1)^2} \end{bmatrix}, \tag{3.72}$$

and $\boldsymbol{Z}$ is a length-N vector of the Fourier transform of $\boldsymbol{x}$. We assume that N is very large, so that the data cannot be stored in one processor. In the beginning, each processor has a segment of consecutive values of the input vector $\boldsymbol{x}$, e.g., Processor 1 has $\begin{bmatrix} x_1 & x_2 & \cdots & x_{N/K} \end{bmatrix}^T$.

3.2.2.2 Preliminaries: Distributed FFT Algorithm

We want to explain the *"transpose"* algorithm that is commonly used in high-performance FFT libraries [36]. It uses the Cooley-Tukey technique to break down N-point FFT into smaller FFTs of size N_1 and N_2 where $N = N_1 N_2$. Now, (3.71) can be rewritten as

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

where $k = k_1 N_2 + k_2,\ k_1 = 0, \cdots N_1 - 1,$ and $k_2 = 0, \cdots, N_2 - 1$. The terms t_{n_1,k_2}'s are called twiddle factor which are equal to $\omega_N^{k_2 n_1}$.

We can now compute N-point FFTs in two steps. In the first step, each processor is assigned to compute N_1/K FFTs of length N_2. Then the processors transpose the data (requiring communication) and compute N_2/K FFTs of size N_1 in the second step. Between the first and the second step, we have to multiply twiddle factors. This complicates our coding approach since

multiplying twiddle factors is an element-wise multiplication of two matrices (Hadamard product), which does not commute with matrix-matrix multiplication (See Remark 3.2.3). We now explain the algorithm in detail:

Algorithm 1. ***Uncoded Distributed FFT Algorithm (Transpose Algorithm)***

1. Rearrange the input data $\boldsymbol{x}$ into X:

$$X = \begin{bmatrix} x_1 & x_{N_1+1} & \cdots & x_{(N_2-1)N_1+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N_1} & x_{2N_1} & \cdots & x_{N_1N_2} \end{bmatrix} = \begin{bmatrix} X_1^{(\text{row})} \\ \vdots \\ X_K^{(\text{row})} \end{bmatrix} = \begin{bmatrix} X_1^{(\text{col})} & \cdots & X_K^{(\text{col})} \end{bmatrix}.$$

   We use $X_i^{(\text{row})}$'s ($X_i^{(\text{col})}$'s) to denote equal-sized submatrices of X divided horizontally (vertically). From our system assumption, in the beginning, the i-th processor has $X_i^{(\text{col})}$ [3]. To begin the distributed FFT computation, we transpose the data distributed over K processors so that the i-th processor can now have $X_i^{(\text{row})}$.

2. Compute N_1/K row-wise FFTs of size N_2 at each processor.

$$Y_i^{(\text{row})} = X_i^{(\text{row})} F_{N_2}$$

3. Transpose the data so that the i-th processor has $Y_i^{(\text{col})}$.

$$Y = \begin{bmatrix} Y_1^{(\text{row})} \\ \vdots \\ Y_K^{(\text{row})} \end{bmatrix} = \begin{bmatrix} Y_1^{(\text{col})} & \cdots & Y_K^{(\text{col})} \end{bmatrix}$$

4. Multiply twiddle factors at each processor.

$$Y_i^{(\text{col})} = T_{N,i}^{(\text{col})} \circ Y_i^{(\text{col})}$$

[3]This assumption is coming from that it is more natural for a processor to store contiguous data without the knowledge that the next computation is going to be FFT. If we assume that processors have row-wise data in the beginning, we can avoid the first transpose step. This does not change the result in Theorem 3.2.5 in scaling sense.

where $\circ$ represents Hadamard product and T_N is a matrix of twiddle factors

$$T_N = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{N_2-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N_1-1} & \cdots & \omega_N^{(N_1-1)(N_2-1)} \end{bmatrix} = \begin{bmatrix} T_{N,1}^{(\text{col})} & \cdots & T_{N,K}^{(\text{col})} \end{bmatrix}.$$

5. Compute N_2/K column-wise FFTs of size N_1 at each processor.

$$Z_i^{(\text{col})} = F_{N_1} Y_i^{(\text{col})}. \tag{3.73}$$

3.2.2.3 Coded FFT Algorithm

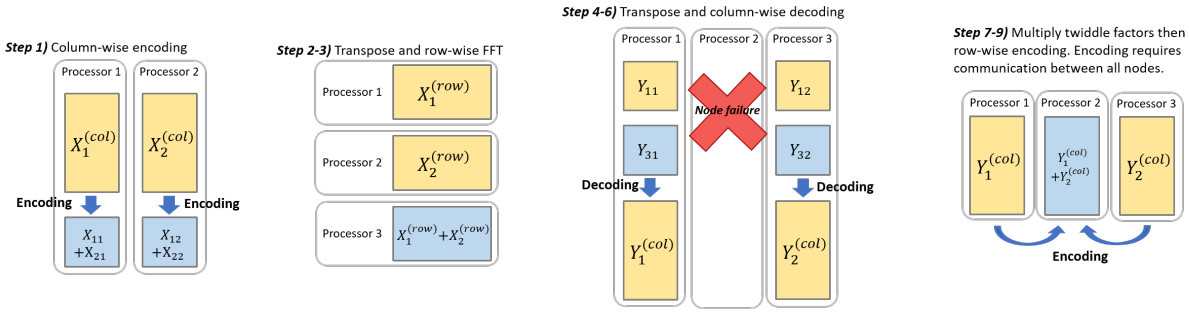


Figure 3.5: This diagram summarizes encoding and decoding steps in Algorithm 2 with an example of $P = 3, K = 2$.

We will now explain our coding strategy for the distributed FFT algorithm. The uncoded distributed algorithm described in Algorithm 1 has transpose step in the middle which requires all the nodes in the system to exchange data with all the other nodes. If there is any failed node before the transpose step, the computation will fail at the transpose step. Hence, simply adding fault tolerance which recovers faults at the end of the algorithm is not adequate for the distributed FFT algorithm. We need to apply fault resilience technique twice: once right before the transpose step, and once when the entire computation is complete. This requires *distributed*

encoding and decoding in the middle of the computation which poses unique challenges for coded FFT algorithm.

In our coding strategy, we utilize $(P-K)$ redundant processors to encode the first and the second FFT steps separately. In the first step, processors perform FFT on the row-wise data $X_i^{(row)}$'s. In order to protect from the lost output at a failed node, we have to encode parity symbols across columns (column-wise encoding). By doing this, at the end of the first step, any successful K processors can recover the output and proceed to the next step. In the second step, each processor computes FFT on the column-wise data, $Y_i^{(col)}$'s, so we encode row-wise parity symbols (row-wise encoding). Our coded computing algorithm is described below (*: additional steps that are not present in the uncoded algorithm).

Algorithm 2. ***Coded Distributed FFT Algorithm***

1. * Encode column-wise parity symbols at each processor.

$$\tilde{X} = G_1^T X = \begin{bmatrix} \tilde{X}_1^{(row)} \\ \vdots \\ \tilde{X}_P^{(row)} \end{bmatrix} \tag{3.74}$$

   G_1 is an N_1-by-N_1' encoding matrix for where $N_1' = \frac{P}{K} N_1$:

$$G_1 = \begin{bmatrix} I_{N_1} & \mathcal{P}_1 \end{bmatrix} \tag{3.75}$$

2. Rearrange the encoded data. Now the i-th processor has $\tilde{X}_i^{(row)}$.
3. Compute N_1/K row-wise FFTs of size N_2 at each processor.
4. Wait for the first successful K processors and transpose the output within the successful K processors.
5. * If needed, decode to retrieve the uncoded output at each processor.
6. Multiply twiddle factors.
7. * Encode row-wise parity symbols and send them to the remaining $P-K$ processors.

$$\tilde{Y} = Y G_2 = \begin{bmatrix} \tilde{Y}_1^{(col)} & \dots & \tilde{Y}_P^{(col)} \end{bmatrix} \tag{3.76}$$

G_2 is an N_2-by-N_2' encoding matrix where $N_2' = \frac{P}{K}N_2$:

$$G_2 = \begin{bmatrix} I_{N_2} & \mathcal{P}_2 \end{bmatrix} \tag{3.77}$$

8. Compute N_2/K row-wise FFTs of size N_1 at each processor.
9. * Wait for the first successful K processors and halt the remaining $P - K$ processors. Decode if needed.

For both encoding steps in Step 1 and Step 7, we use *a (P, K) systematic MDS code*. In the following theorem, we show that using the proposed coded distributed FFT algorithm, any K successful processors are enough to recover the computed outputs at Step 5 and Step 9 [4].

Theorem 3.2.4. *In Algorithm 2 where we compute distributed FFT of size N using P processors each of which can store and process $\frac{1}{K}$ fraction of the input ($P > K$), any successful K processors can recover Y and Z at Step 5 and 9, respectively.*

Proof. Let us first prove that we can recover Y with any K successful processors at Step 5 and the similar argument holds for recovering Z at step 9.

At Step 4, we will have the result from K successful workers. Let us denote the indices of the successful K workers as $\{i_1, \cdots, i_K\}$. Then the output from the successful workers is:

$$Y_{\text{suc}} = \begin{bmatrix} \tilde{X}^{(\text{row})}_{i_1} F_{N_2} \\ \tilde{X}^{(\text{row})}_{i_2} F_{N_2} \\ \vdots \\ \tilde{X}^{(\text{row})}_{i_K} F_{N_2} \end{bmatrix} = \begin{bmatrix} Y^{(\text{col})}_{\text{suc},i_1} & \cdots Y^{(\text{col})}_{\text{suc},i_K} \end{bmatrix}. \tag{3.78}$$

After transposing at Step 5, processors $i_1, \cdots, i_K$ will have column-wise output $Y^{(\text{col})}_{\text{suc},i_1}, \cdots Y^{(\text{col})}_{\text{suc},i_K}$. $Y^{(\text{col})}_{\text{suc},i}$ can be written as:

$$Y^{(\text{col})}_{\text{suc},i} = G^T_{1,suc} X F^{(\text{col})}_{N_2,i} = G^T_{1,suc} Y^{(\text{col})}_i \tag{3.79}$$

[4]Note that we do not have any fault recovery for twiddle multiplication step. However, computational complexity of twiddle factor multiplication is $O(N)$ compared to that of $O(N \log N)$. Hence, it is less probable to have faults during twiddle factor multiplication step

where $G_{1,suc}^T$ is a submatrix of G_1^T which only has rows from successful nodes and hence has the size N_1-by-N_1.

As we assume the erasure model where we lose the entire data from a failed node, we only code across nodes, not within a node. Hence, our encoding matrix G_1 has the following structure:

$$G_1 = \mathcal{G}_1 \otimes I_{N_1/K} \tag{3.80}$$

where $\mathcal{G}_1$ is the encoding matrix for a systematic (P, K)-MDS code which has size K-by-P.

Now, $G_{1,\text{suc}}$ can be rewritten as:

$$G_{1,\text{suc}}^T = \mathcal{G}_{1,\text{suc}}^T \otimes I_{N_1/K} \tag{3.81}$$

where $\mathcal{G}_{1,\text{suc}}$ is a submatrix of $\mathcal{G}$ that only has K columns from the K successful nodes, i.e., i_1-th to i_K-th columns of $\mathcal{G}$. Because $\mathcal{G}_1$ is a (P, K) MDS code, $\mathcal{G}_{1,\text{suc}}$ always has a full rank. As $\text{rank}(A \otimes B) = \text{rank}(A) \cdot \text{rank}(B)$ for any matrices A and B, $\text{rank}(G_{1,\text{suc}}) = N_1$. Hence, we can recover $Y_i^{(\text{col})}$ at every successful node at Step 5. Similar argument applies to recovering Z at Step 9. ■

3.2.2.4 Communication Cost Analysis

Now, we prove our main theorem which states that as long as the number of parity processors is $o(\log K)$, communication overhead of encoding and decoding can be amortized:

Theorem 3.2.5. *In our proposed coded FFT algorithm, if $P - K = o(\log_2 K)$, communication overhead of coding is negligible compared to the communication cost of uncoded FFT.*

To prove the theorem, we first identify the communication cost of uncoded FFT algorithm. Then, we analyze communication cost of encoding and decoding and we compare them to obtain the theorem.

Communication cost of uncoded FFT algorithm. Let us begin with understanding the communication cost of uncoded FFT algorithm. In Algorithm 1, steps that require communication

are Step 1 and 3. Both steps need communication to transpose the data stored in distributed processors. For transposing the data, all processors have to exchange data with all the other processors. This communication is known as *"all-to-all"* communication. Bruck et al. showed lower bounds and explicit algorithms that achieve lower bounds for two special cases of all-to-all communication [11] – a minimum-communication-rounds regime and a minimum-bandwidth regime. Let us first formally define all-to-all communication.

Definition 3.2.1. [All-to-all] In all-to-all(p, n) communication, there are p nodes each of which stores n symbols. The data stored in the i-th node can be broken down into p data blocks, $M_{i,1}, \cdots M_{i,p}$, where the size of each block is n/p symbols. The goal of the communication is to transpose the data stored in p processors so that at the end of the communication, the i-th node has $M_{1,i}, \cdots, M_{p,i}$ data blocks.

We will first give a simple lower bound of all-to-all(p, n) communication.

Theorem 3.2.6 (Proposition 2.3 and 2.4 in [11])**.** *For all-to-all(p, n) communication, C_1 and C_2 are lower bounded by:*

$$C_1 \geqslant \lceil \log_2 p \rceil, \quad C_2 \geqslant \frac{p-1}{p} n \tag{3.82}$$

However, Bruck et al. showed that the lower bounds on C_1 and C_2 cannot be achieved simultaneously which is stated in the theorem below.

Theorem 3.2.7 (Theorem 2.5 and 2.6 in [11])**.** *If all-to-all(p, n) communication uses the minimum number of rounds, i.e., $C_1 = \lceil \log_2 p \rceil$, C_2 is lower bounded by:*

$$C_2 \geqslant \frac{n}{2} \log_2 p. \tag{3.83}$$

If all-to-all(p, n) communication uses the minimum number of symbols transferred in sequence, i.e., $C_2 = \frac{p-1}{p} n$ symbols in a sequence, then C_1 is lower bounded by:

$$C_1 \geqslant p - 1. \tag{3.84}$$

Furthermore, both lower bounds are achievable.

Now, by using Theorem 3.2.7, we can give communication cost lower bounds on the transpose step in the distributed FFT algorithm.

Corollary 3.2.1. *The transpose step of N-point FFT requires the communication cost at least*

$$\lceil \log_2 K \rceil \alpha + \frac{1}{2} \frac{N}{K} \log_2 K \beta \tag{3.85}$$

when using the minimum communication rounds regime, and

$$(K-1)\alpha + \frac{(K-1)}{K} \frac{N}{K} \beta \tag{3.86}$$

when using the minimum communication bits regime.

Under our massively parallel system model where K is very large, we have $\log K << \sqrt{K}$. Hence, we should always choose the minimum-communication-round regime over the minimum-bandwidth regime. From now on, *we will only consider minimum communication round regime* and use its communication cost given in (3.85).

Communication overhead of coding. Now, let us identify additional communication cost due to coding in Algorithm 2. In the first encoding step where we compute column-wise parity symbols, we do not need any communication since processors already have column-wise data in the beginning. Also, for the first decoding in Step 5, column-wise decoding can be done in local processors as each processor has column-wise data after the transpose step. In Step 7, it requires inter-processor communication to encode row-wise parity symbols as one row of the data is spread over all the processors. Also in step 9, we have to perform row-wise decoding while every node has column-wise data, and thus we need inter-processor communication for decoding. Hence, in this section, *we will analyze the communication cost of the second encoding step and decoding step*. We will first show the communication cost of the second encoding step where we compute:

$$\tilde{Y} = YG_2. \tag{3.87}$$

Before we begin our communication cost analysis, we want to make a few remarks.

Remark 3.2.3. [Why do we need distributed encoding?] If we can do the second encoding, which is computing row-wise parity symbols, at local processors before the transpose step, we can avoid communication for distributed encoding at Step 7. However, there is no trivial way of doing this using a linear code due to the twiddle factors. After Step 3, the i-th processor has

$$Y_i^{(\text{row})} = \tilde{X}_i^{(\text{row})} F_{N_1} = G_{1,i}^{(\text{row})} X F_{N_1}. \tag{3.88}$$

If we do row-wise encoding at the i-th processor locally before the transpose step, the i-th processor will have

$$\tilde{Y}_i^{(\text{row})} = G_{1,i}^{(\text{row})} X F_{N_2} G_2. \tag{3.89}$$

We then perform the transpose of the output from the first K successful nodes. The i-th node now has

$$\tilde{Y}_i^{(\text{col})} = G_{1,suc} X F_{N_2} G_{2,i}^{\text{col}}. \tag{3.90}$$

Column-wise decoding can be done locally by inverting $G_{1,suc}$:

$$\hat{Y}_i^{(\text{col})} = G_{1,suc}^{-1} G_{1,suc} X F_{N_2} G_{2,i}^{\text{col}} = X F_{N_2} G_{2,i}^{\text{col}}. \tag{3.91}$$

We now have to multiply twiddle factors to $\hat{Y}_i^{(\text{col})}$:

$$\hat{Y}_i^{(\text{col})} = T_N \circ \hat{Y}_i^{(\text{col})} = T_N \circ (X F_{N_2} G_{2,i}^{\text{col}}) \tag{3.92}$$

However, this will produce a different final output from what we expect because of the nonlinearity of Hadamard product:

$$A \circ (BC) \neq (A \circ B)C. \tag{3.93}$$

Hence,

$$T_{N,i}^{(col)} \circ (X F_{N_2} G_{2,i}^{\text{col}}) \neq (T_{N,i}^{(col)} \circ X F_{N_2}) G_{2,i}^{\text{col}}. \tag{3.94}$$

From our modified coding strategy, our final output from successful nodes will be $F_{N_1} T_N \circ (X F_{N_2} G_{2,\text{suc}})$ and even after decoding, we will have

$$F_{N_1} T_N \circ (X F_{N_2} G_{2,\text{suc}}) G_{2,\text{suc}}^{-1} \neq F_{N_1} T_N \circ (X F_{N_2}). \tag{3.95}$$

This means that we have to perform twiddle factor multiplication before proceeding to the row-wise encoding step. With the same argument, we can show that column-wise decoding must be done before multiplying twiddle factors. It concludes that because of the twiddle factors, the second-step encoding must be done across the processors incurring some communication cost.

We now want to analyze the communication cost of the second encoding step. Let us first investigate the communication cost of a simple encoding scheme where we add one parity node that stores the checksums of data, $X_1 + \cdots + X_K$. The encoding matrix G_2 for this can be written as follows:

$$\mathcal{G}_{\text{cks}} = \begin{bmatrix} & & 1 \\ & I_K & \vdots \\ & & 1 \end{bmatrix} \tag{3.96}$$

$$G_2 = \mathcal{G}_{\text{cks}} \otimes I_{N_2/K} \tag{3.97}$$

For this computation, all K nodes have to send its data to one checksum node to compute the sum of all the data in the network. This is a well-known communication operation called *"reduce(-to-one)"*.

Definition 3.2.2. [Reduce] In reduce(p, n) communication, there are p data nodes which have data $M_1, \cdots, M_p$ of size n and one reduction node. The goal of the communication is to send $M_1 + \cdots + M_p$ to the reduction node.

A lower bound on the communication cost of reduce(p, n) operation is given in the following theorem.

Theorem 3.2.8. *The communication cost of reduce(p, n) is lower bounded by*

$$\lceil \log_2 p \rceil \alpha + n\beta. \tag{3.98}$$

It was found that reduce operation can be done by reversing any broadcasting algorithm, where one broadcasting node sends its message to all the other processors in the network. Traff and Ripke[111] proposed a near-optimal broadcasting algorithm that achieves the lower bound

(3.98) within a factor of 2 . By reversing their broadcasting algorithm, we can achieve the same communication cost for reduce(p, n) communication.

Theorem 3.2.9. *Reduce(p, n) can be done with the communication cost of at most*

$$(\sqrt{\lceil \log_2 p \rceil \alpha} + \sqrt{n\beta})^2 \leqslant 2(\lceil \log_2 p \rceil \alpha + n\beta). \tag{3.99}$$

Whether (3.99) is optimal or not is an open problem. We will use this as a state-of-the-art communication algorithm for reduce operation. By applying (3.99), we can obtain the communication cost for encoding one checksum node.

Corollary 3.2.2. *A $(K + 1, K, 2)$ systematic MDS code over K systematic processors each of which hs N/K data symbols can be encoded with the communication cost of*

$$(\sqrt{\lceil \log_2 K \rceil \alpha} + \sqrt{N/K\beta})^2 \leqslant 2(\lceil \log_2 K \rceil \alpha + N/K\beta). \tag{3.100}$$

We can now extend computing checksums to computing parity symbols for a generic $(P, K, d = P - K + 1)$ systematic MDS code. Unlike checksum computation which only requires a single reduce(-to-one) operation, here we need multiple reductions to $P - K$ nodes.

From the intuition we got from reduce(-to-one) problem, we will first establish bounds for multi-broadcasting problem (will be defined below) and show that multi-reduce problem for encoding a $(P, K, d = P - K + 1)$ systematic MDS code can be solved by reversing the multi-broadcasting algorithm.

Definition 3.2.3. [Multi-broadcast] In multi-broadcast(p, r, n) communication, there are r broadcasting nodes and p destination nodes. Broadcasting nodes have distinct messages $M_1, \cdots, M_r$ of size n symbols. At the end of the communication, all p destination nodes should have all r messages, $M_1, \cdots, M_r$.

We want to note that multi-message broadcasting has been studied in the literature [4, 98]. However, their models have one broadcasting node which sends multiple messages in a sequence. This is fundamentally different from our *multi-broadcast* which has multiple broadcasting nodes

that can send out their messages simultaneously. To the best of our knowledge, communication cost analysis of this specific problem has not been studied before.

We will first show a communication algorithm for multi-broadcast(p, r, n) and then show that it achieves the lower bound within a factor of 2.

Theorem 3.2.10. *Multi-broadcast*(p, r, n) *can be done with the communication cost at most*

$$2(\lceil \log_2 p \rceil \alpha + rn\beta) \tag{3.101}$$

Proof. First, divide p processors into r disjoint sets of size p/r. Let us denote the sets as $S_1, S_2, \cdots, S_r$. The i-th broadcasting node broadcasts its message to all the nodes in S_i. With the optimal broadcasting algorithm [111], it takes communication cost of $(\sqrt{\log_2 \frac{p}{r}\alpha} + \sqrt{n\beta})^2$.

After the broadcasting step, the j-th nodes in S_i's $(i = 1, \cdots, r)$ communicate with each other so that all of them can share $M_1, \cdots, M_r$. This is all-gather(r, n) communication which is defined as follows.

Definition 3.2.4. [All-gather] In all-gather(p, n) communication, there are p nodes which have distinct messages $M_1, \cdots, M_p$ of size n symbols. At the end of the communication, all p nodes should have all p messages.

All-gather(r, n) can be done with communication cost of $(\log_2 r)\alpha + (r-1)n\beta$ using the *bidirectional algorithm* [14].

The total communication cost of this two-step algorithm is

$$\begin{aligned}
&(\sqrt{\log_2 \frac{p}{r}\alpha} + \sqrt{n\beta})^2 + \log_2 r\alpha + (r-1)n\beta \\
&\leqslant \lceil \log_2 p \rceil \alpha + rn\beta + (\log_2 \frac{p}{r}\alpha + n\beta) \\
&\leqslant 2(\lceil \log_2 p \rceil \alpha + rn\beta).
\end{aligned}$$

■

We now show a lower bound for multi-broadcast(p, r, n) communication.

Theorem 3.2.11. *The communication cost of multi-broadcast*(p, r, n) *is lower bounded by*

$$\lceil \log_2 p \rceil \alpha + rn\beta \tag{3.102}$$

Proof. Each broadcasting node must communicate to p destination nodes which takes at least $\lceil \log_2 p \rceil$ communication rounds. Each destination node has to receive messages $M_1, \cdots M_r$ which have n. Hence, mutlti-broadcast(p, r, n) requires at least the bandwidth of rn. ■

By comparing (3.101) and (3.102), we can see that the algorithm given in Theorem 3.2.10 achieves the lower bound within a factor of 2.

Finally, we define *multi-reduce* operation which is the communication required for encoding parity symbols, and show that it can be done with the same communication cost as multi-broadcast operation.

Definition 3.2.5. [Multi-reduce] In multi-broadcast(p, r, n) communication, there are p data nodes and r reduction nodes ($r < p$). p data nodes have data $M_1, \cdots, M_p$ each of which consist of n symbols. At the end of communication, the i-th reduction node will have $a_{i,1}M_1 + \cdots + a_{i,p}M_p$ where $a_{i,j}$'s ($i = 1, \cdots, r, j = 1, \cdots, p$) are chosen so that the data from any p nodes are linearly independent combinations of $M_1, \cdots, M_p$.

Theorem 3.2.12. *Multi-reduce*(p, r, n) *communication can be done by reversing the multi-broadcast algorithm given in Theorem 3.2.10. Hence, the communication cost of multi-reduce*(p, r, n) *is at most*

$$2(\lceil \log_2 p \rceil \alpha + rn\beta) \tag{3.103}$$

Proof. Let $D_1, D_2, \cdots, D_p$ denote the data at p data processors. Let us divide data processors into r disjoint sets of size p/r and let S_i denote the set of indices of the i-th set: $S_i = \{(i-1) \cdot p/r + 1, \cdots, (i-1) \cdot p/r + p/r\}$. This is all-gather$(r, n)$ communication.

First, the j-th nodes in S_i's ($i = 1, \cdots, n$) perform all-gather communication. All the j-th processors in S_i's will have $D_j, D_{j+p/r}, \cdots, D_{j+(r-1)p/r}$ after the communication.

In the second step, all the nodes in S_i will carry out reduce communication with the i-th reduction node. Each node in S_i will compute a corresponding linear combination of the the data it has and send only n symbols of data to the i-th reduction node. For instance, the j-th node in S_i will compute

$$a_{i,j}D_j + a_{i,j+p/r}D_{j+p/r} + \cdots + a_{i,j+(r-1)p/r}D_{j+(r-1)p/r}.$$

This is reduce$(p/r, n)$ which can be done with the communication cost of $(\sqrt{\log_2 \frac{p}{r}\alpha} + \sqrt{n\beta})^2$. This completes multi-reduce(p, r, n) communication. ■

This gives an achievable communication scheme for encoding parity symbols and decoding systematic symbols of a (P, K, d) systematic MDS code.

Corollary 3.2.3. *A $(P, K, d = P - K + 1)$ systematic MDS code over K systematic processors each of which has N/K data symbols can be encoded with the communication cost of*

$$2\left(\lceil \log_2 K \rceil \alpha + (P - K)\frac{N}{K}\beta\right). \tag{3.104}$$

Proof. The encoding matrix of $(P, K, P - K + 1)$ MDS code has the form

$$\mathcal{G} = \left[\begin{array}{c|c} I_K & \mathcal{P} \end{array}\right]$$

where I_K is a K-by-K identity matrix and $\mathcal{P}$ is a parity matrix of dimension K-by-$P - K$ whose entries are all non-zero [7]. This means that every parity symbol is a linear combination of all K symbols distributed in K nodes. Hence, encoding parity symbols for a systematic $(P, K, d = P - K + 1)$ MDS code is exactly multi-reduce$(K, P - K, N/K)$ operation. Simply substituting this to (3.103) completes the proof. ■

A similar argument can be applied to show that decoding at Step 11 of Algorithm 2 can also be done with the same communication cost.

Corollary 3.2.4. *Reconstructing N/K data symbols in failed systematic nodes of at Step 11 of Algorithm 2 can be done with the communication cost at most:*

$$2\left(\lceil \log_2 K \rceil \alpha + (P - K)\frac{N}{K}\beta\right). \tag{3.105}$$

Proof. First, note that we only have to recover the data in systematic nodes. The worst case is when there are $P-K$ failed nodes among the systematic nodes. In this case, the remaining K successful nodes have to send their data to $P-K$ systematic nodes. A failed node's data symbol can be represented as a linear combination of K output symbols from successful nodes. Hence, this is multi-reduce$(K, P-K, N/K)$ operation. ■

Proof. (Proof of of Theorem 3.2.5.) By comparing the encoding communication overhead given in (3.104) with the communication cost of uncoded FFT algorithm given in (3.85), we can prove our main theorem. Uncoded FFT algorithm requires two transpose operation, one in the beginning and one before the second FFT step. This requires communication cost of

$$2\left(\lceil \log_2 K \rceil \alpha + \frac{N}{2K} \lceil \log_2 K \rceil \beta\right) \tag{3.106}$$

If we compare this against the communication cost of encoding given in (3.104), the condition for the encoding cost to be smaller than the all-to-all communication is given as follows:

$$4\big(\lceil \log_2 K \rceil \alpha + (P-K)\frac{N}{K}\beta\big) < 2\big(\lceil \log_2 K \rceil \alpha + \frac{N}{2K} \lceil \log_2 K \rceil \beta\big)$$
$$P - K < \frac{\log_2 K}{4}.$$

Hence, as long as $P-K$ is smaller than $\frac{\log_2 K}{4}$ in scaling sense, communication overhead of coding is negligible compared to the intrinsic communication cost of uncoded distributed FFT algorithm. ■

Algorithm 2 2.5D SUMMA

Initial Data Distribution: $P(i, j, 1)$ has $\mathbf{A}_{i,j}$ and $\mathbf{B}_{i,j}$.

/* Distributing $\mathbf{A}$, $\mathbf{B}$ across layers

for $k = 1$ **to** c **do**

  for $i = 1$ **to** $\sqrt{p/c}$ **do**

    for $j = 1$ **to** $\frac{1}{c}\sqrt{p/c}$ **do**

      /* All $P(i, j, k)$ in parallel */

      $P(i, j, 1)$ sends $\mathbf{A}_{i,j}$ and $\mathbf{B}_{i,j}$ to $P(i, j, k)$

    end for

  end for

end for

for $k = 1$ **to** c **do**

  /* All k-th layers in parallel */

  Perform SUMMA to compute $\mathbf{A}_k^{\text{col}}\mathbf{B}_k^{\text{row}}$.

end for

for $i = 1$ **to** $\sqrt{p/c}$ **do**

  for $j = 1$ **to** $\sqrt{p/c}$ **do**

    /* All i, j in parallel */

    $P(i, j, 1), \ldots P(i, j, c)$ reduce to $P(i, j, 1)$ to compute $\mathbf{C}_{i,j}$.

  end for

end for

Algorithm 3 Decoding for (n, k) coded SUMMA

while (# decoded rows) $< k$ or (# decoded cols) $< k$ **do**

for $i = 1$ **to** n **do**

if Row i has $\geqslant k$ successful nodes **then**

Perform `row-decode`

end if

end for

for $j = 1$ **to** n **do**

if Column j has $\geqslant k$ successful nodes **then**

Perform `column-decode`

end if

end for

end while

function `row-decode` (i)

Decoding set $\mathcal{D} = \{j_1, \cdots, j_k\}$

`all-to-all`($\mathcal{D}$)

/* For all nodes in $\mathcal{D}$ in parallel */

Locally decode for $\mathbf{C}_{i,j}$, $(j = 1, \cdots, k)$

end function

function `col-decode` (j)

Decoding set $\mathcal{D} = \{i_1, \cdots, i_k\}$

`all-to-all`($\mathcal{D}$)

/* For all nodes in $\mathcal{D}$ in parallel */

Locally decode for $\mathbf{C}_{i,j}$, $(i = 1, \cdots, k)$

end function

Algorithm 4 Coded SUMMA

Input: Input matrices $\mathbf{A}$ and $\mathbf{B}$, generator matrix $\mathbf{G}_{k\times n}$.

Initialize (Encoding): Partition matrix $\mathbf{A}$ into k row blocks and $\mathbf{B}$ into k column blocks. Encode the k row blocks of $\mathbf{A}$ into n row blocks. Encode the k column blocks of $\mathbf{B}$ into n column blocks. Denote the encoded matrices by $\mathbf{A}_{\text{coded}}$ and $\mathbf{B}_{\text{coded}}$.

Initialize (Data distribution): Suppose there is a $n \times n$ processor mesh. Partition the coded matrix $\mathbf{A}_{\text{coded}}$ into $n \times n$ square blocks and $\mathbf{B}_{\text{coded}}$ into n square blocks. Send each $\mathbf{A}_{\text{coded,ij}}$ and $\mathbf{B}_{\text{coded,ij}}$ to the processor on the i-th row and the j-th column. Initialize $\mathbf{C}_{\text{coded,ij}} = 0$;

Multi-stage Computing: The matrix multiplication of $\mathbf{C}_{\text{coded}} = \mathbf{A}_{\text{coded}}\mathbf{B}_{\text{coded}}$ is computed using n outer-product stages, i.e.,

$$\mathbf{A}_{\text{coded}}\mathbf{B}_{\text{coded}} = \sum_{l=1}^{n} \mathbf{A}_{\text{coded,l}}\mathbf{B}_{\text{coded,l}}. \tag{3.69}$$

for $l = 1$ **to** n **do**

for $i = 1$ **to** n, $j = 1$ **to** n (in parallel) **do**

The il-th processor broadcasts its $\mathbf{A}_{\text{coded,il}}$ to the other processors in the i-th row.

The lj-th processor broadcasts its $\mathbf{B}_{\text{coded,lj}}$ to the other processors in the j-th column.

The ij-th processor should receive $\mathbf{A}_{\text{coded,il}}$ and $\mathbf{B}_{\text{coded,lj}}$. Then, it computes

$$\mathbf{C}_{\text{coded,ij}} \mathrel{+}= \mathbf{A}_{\text{coded,il}}\mathbf{B}_{\text{coded,lj}}. \tag{3.70}$$

if The number of faults in a row or column is above a threshold **then**

Conduct decoding in the entire mesh, on both rows and columns, in parallel.

end if

end for

end for

Chapter 4

Experimental Results

In this chapter, we present experimental evaluation of coded computing strategies. In Section 4.1, we show implementation and experiments of Coded SUMMA strategy (Section 3.2.1) on a HPC cluster.

4.1 Coded SUMMA Experiments

In this section we compliment the theoretical analysis on its fault tolerance and execution time overhead given in Section 3.2.1 with extensive experimental evaluations.

Experimental Setup

In our experimental setup, we used a cluster with 40 compute nodes, each of which has two 12-Core AMD Opteron (tm) Processor 6164 HE, 64 GB DRAM, and 500 GB hard disk. Nodes are connected through Gigabit Ethernet under a single switch. We used each core as one MPI process, *i.e.,* one core was one logical node $P(i, j, l)$. To ensure that there is no MPI communication within the same compute node, we used cyclic distribution of compute nodes. We injected a layer failure by artificially ignoring the result from one layer in the reduce phase. We assumed that the information about the failed node will be made available at all surviving nodes. We

recorded execution time of: memory allocation, MatDot Encoding, MPI Scatter, 2D SUMMA, and Decoding + MPI Reduce. In the implementation of the baseline replication-based scheme, time spent in the communication from the surviving replica layer to the first layer was included in Decoding + MPI Reduce time. Since the cluster we used for experiments had total of 960 cores, the most extensive experiments were run on an $8 \times 8 \times 4$ grid with total of 256 cores[1].

Table 4.1: Execution time comparison of $(n = 8, m = 2, M = 4)$ 3D Coded SUMMA and replication. We used systematic MatDot codes and 8 cores per node.

N	Strategy	Memory Allocation (s)	Encoding (s)	Scatter (s)	2D SUMMA (s)	Decoding + Reduce (s)	Total (s)
10000	Replication	0.1	0	1.505	19.583	0.926	22.245
	MatDot	0.105	0.124	2.25	18.621	1.384	22.486
20000	Replication	0.369	0	6.574	87.792	3.626	98.681
	MatDot	0.362	0.402	9.075	88.371	5.502	103.357
30000	Replication	0.75	0	14.993	214.798	7.859	239.035
	MatDot	0.752	0.864	19.773	224.232	12.316	257.883
40000	Replication	1.317	0	25.613	438.356	13.941	480.464
	MatDot	1.325	1.418	39.496	440.872	21.853	505.41

Comparison with replication

We first compare our proposed MatDot-coded approach and replication. Execution time comparison of the two is summarized in Table 4.1. First notice that almost 90 % of the total execution time is used on 2D SUMMA. Then, the next significant portion of the execution time is MPI Scatter and Reduce. Computation time for MatDot encoding and decoding makes up less than

[1]Bigger grids with the dimensions of non-power-of-two numbers are not included as they showed worse performance.

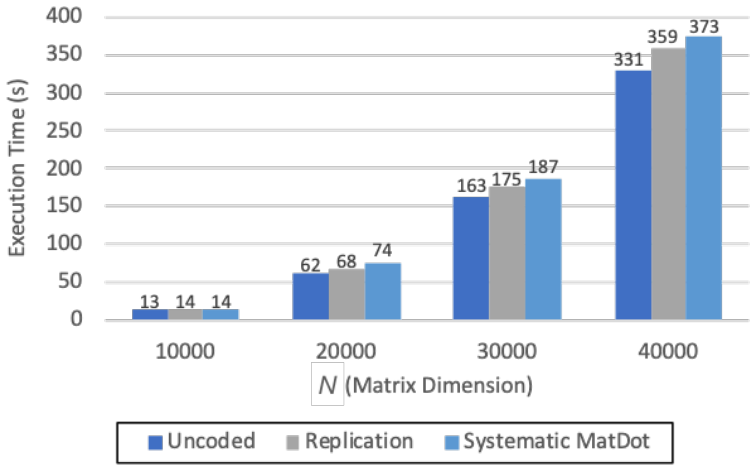
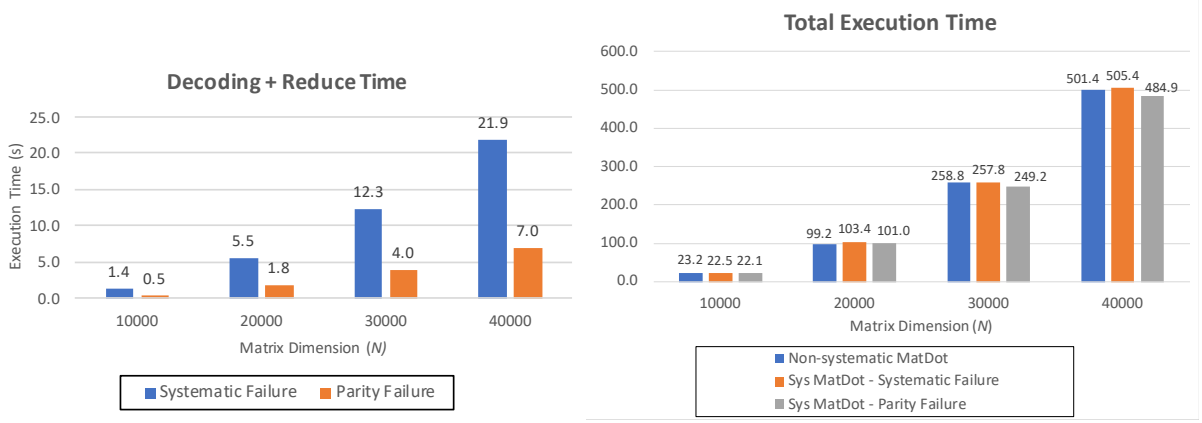


Figure 4.1: Comparison of total execution time for uncoded 3D SUMMA with no redundancy, replication, and coded 3D SUMMA using systematic MatDot codes. We can see that the overhead of the coded strategy is about 5-7% compared to replication and 10-18% compared to uncoded.

1 % of the total time. When we compare the total execution time, the overhead of MatDot coding is about 5-7 % compared to replication. This is mainly due to the increased communication cost in the scatter and reduce communication as predicted in the previous section. We further compare the total execution time of replication and MatDot against the uncoded counterpart that does not provide any resilience (See Fig. 4.1). Compared to the uncoded strategy, the execution time of 3D Coded SUMMA is about 10-18 % more.

Fig. 4.2 shows the difference between using systematic and non-systematic codes. In Fig. 4.2a, systematic failure means a node failure in a systematic layer (the first m layers with the original data) and parity failure means a node failure in a parity layer (the last m layers with encoded data). The biggest benefit of using systematic codes is that when there is no failure in systematic nodes, there is no need for decoding, and the final steps would be no different from the uncoded strategy. The results in Fig. 4.2a show that this is indeed true in experiments and the last reduce step (including decoding) is about 3x times faster when we have only parity failures, and no



(a) Comparison of decoding+reduce time using Systematic MatDot codes.

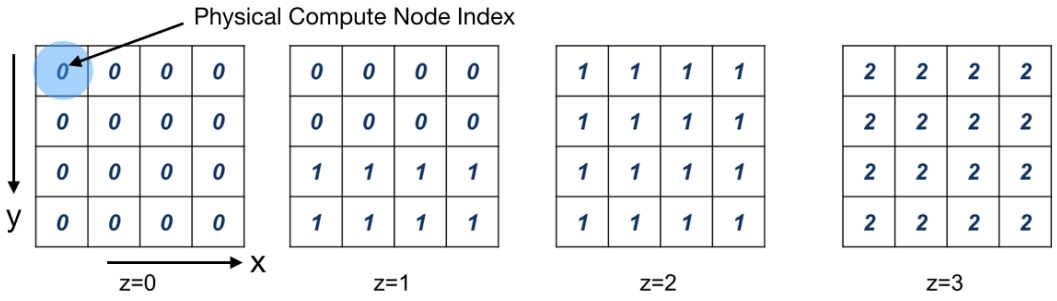
(b) Comparison of total execution time for using non-systematic MatDot codes and systematic MatDot codes.

Figure 4.2: (a) When the failed node is a parity node, there is no need for decoding, and hence reducing over the first m systematic nodes is sufficient. This reduces decoding+reduce time by $\sim$3x. (b) For systematic MatDot codes, we include both systematic failure and parity failure cases in the comparison. For systematic failures, non-systematic and systematic codes share similar performance. For parity failures, systematic codes show clear advantage when the matrix dimension is large.

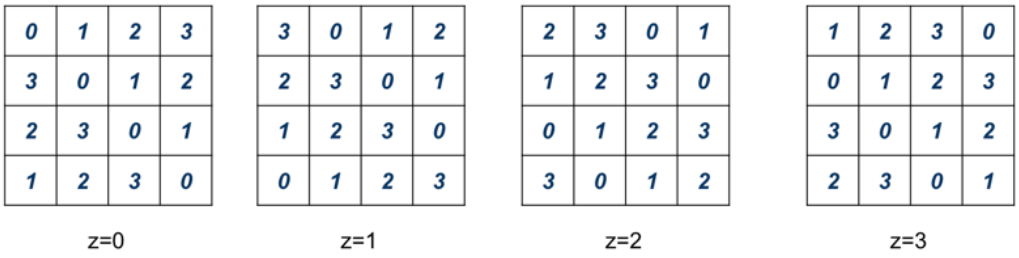
systematic failure. Because of this effect, we can see that using systematic codes is about 3-5 % faster than non-systematic codes when there is no systematic failure in Fig. 4.2b.

Master-Worker vs. Masterless

We now demonstrate interesting side results that we obtained through our experiments on Coded 2.5D SUMMA. Recall that in the results presented above used the *elemental cyclic distribution* of physical nodes as given in Figure 4.3b. However, the initial implementation followed a brute-force node distribution as given in Figure 4.3a. In the brute-force approach, the first physical node (node index 0 in Figure 4.3) takes up all logical nodes on the first layer ($z = 0$). In Coded 2.5D SUMMA, the first layer is responsible for data encoding, distribution, aggregation, and



(a) Brute-force node distribution (Master-worker).

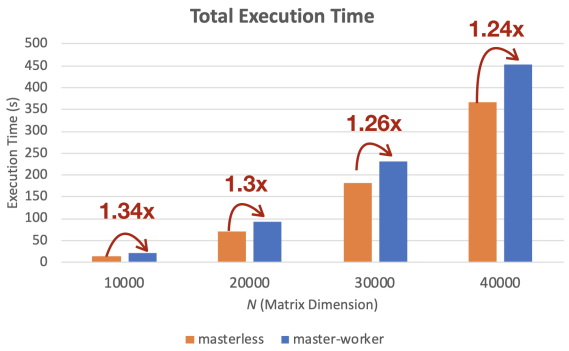


(b) Elemental cyclic node distribution (Masterless).

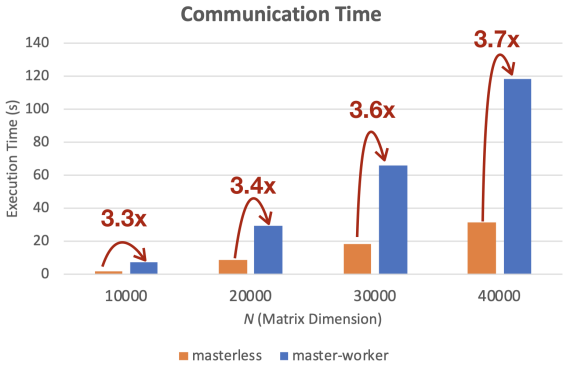
Figure 4.3: An example of the physical compute node distribution for a $4 \times 4 \times 4$ grid. The 3D grid is unrolled in the z-dimension and blue number on the grid represents the physical node index.

decoding, which are tasks assigned to a master node in the master-worker setup considered in coded computing literature [29, 67, 130]. Hence, in this approach, although the logical algorithm is fully-distributed, the computation and communication pattern in the actual implementation works as if one physical node acts as a master node. On the other hand, in the elemental cyclic distribution approach, logical nodes on the first layer are evenly distributed among four physical nodes (node index 0 to 3 in Figure 4.3b). By comparing these two implementations, we analyze the effect of having a master node and provide an experimental proof on why we need masterless strategies.

The comparison of the two implementations is summarized in Figure 4.4. This experiment was also run on an $8 \times 8 \times 4$ grid. From Figure 4.4a, we can see that the total execution time of the master-worker implementation is about 25-35 % higher than the masterless counterpart.



(a) Total execution time comparison.

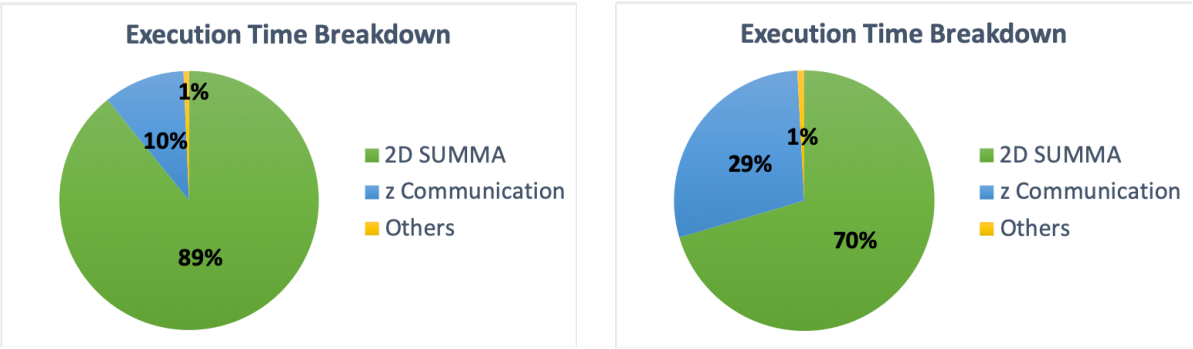


(b) Communication time comparison.

Figure 4.4: Execution time comparison of master-worker and masterless implementations of Coded 2.5D SUMMA for $(n = 8, m = 2, M = 4)$.

Furthermore, Figure 4.4b shows that the increase in communication time is even more dramatic; the master-worker implementation consumes >3x time in communication. This substantiates our claim that communication between a master node and worker nodes will become a significant bottleneck in the parallel algorithm.

Figure 4.5 portrays the breakdown of total execution time in the master-worker and masterless implementations. First, recall that the only communication that is affected by these two different implementations is the z-communications, *i.e.,* z-scatter and z-reduce across layers. The broadcast operations along the x and y axes during 2D SUMMA are not affected. Now, we can notice that in the masterless Coded 2.5D SUMMA implementation, z-communication (master-worker communication) is only about 10 % of the total execution time. In the master-worker implementation, this becomes ∼30 % of the total time. In parallel algorithms that have a higher portion of communication time (e.g., 30 % of total execution time), the increase in communication time due to the existence of a master node would be more severe. Also, the scale of experiments we ran was relatively small with a total of 16 physical nodes. Once we use hundreds of physical nodes, the bottleneck of a master-worker communication would be more evident.



(a) Masterless

(b) Master-Worker

Figure 4.5: Execution time breakdown of the masterless and master-worker implementations of Coded 2.5D SUMMA for $(n = 8, m = 2, M = 4, N = 30000)$.

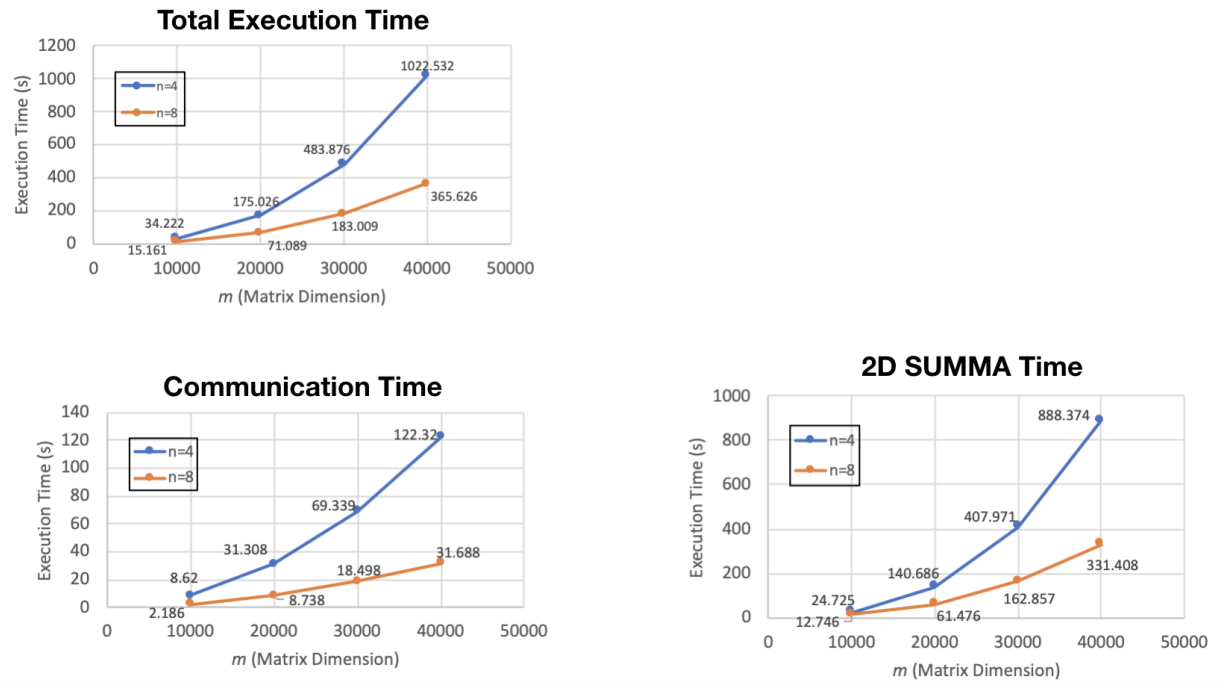


Figure 4.6: Comparing Coded 2.5D SUMMA execution time for $(n = 8, m = 2, M = 4)$ and $(n = 4, m = 2, M = 4)$. Comparing these two different settings suggests that the execution time analysis in Section 3.2.1 is fairly accurate. From $n = 4$ to $n = 8$, we can see that communication cost (z-scatter and z-reduce) reduces by 4x $(1/n^2)$ as expected. On the other hand, 2D SUMMA time reduces by 2-2.7x. This can also be explained by our analysis because computation time in 2D SUMMA is expected to reduce by $1/n^2$ (*i.e.,* by 4x) but communication time in 2D SUMMA is expected to reduce by $1/n$ (*i.e.,* by 2x). Hence, all in all, theoretical analysis suggests an execution time reduction between 2-4x for the 2D SUMMA part.

Appendix A

n-matrix Multiplication

Proof of Theorem A.4.1. Here, we only derive the proof for the case of even n. The proof for odd n can be derived in a similar manner with minor differences in the expressions. What we have to show to complete the proof are as follows:

Claim A.0.1. *The maximum degree of* $p_{\mathbf{C}}(x)$ *is* $s^{\frac{n}{2}}t^{\frac{n}{2}+1} + s^{\frac{n}{2}}t^{\frac{n}{2}-1} - 1$.

Claim A.0.2. $\mathbf{C}_{i,j}$ *is the coefficient of* $x^{d(n,i,j)}$ *for* $i, j = 1, \cdots, t$ *where*

$$d(n,i,j) = s - 1 + s(t-1) + st(s-1) + \cdots + i \cdot s^{\frac{n}{2}}t^{\frac{n}{2}-1} + j \cdot s^{\frac{n}{2}}t^{\frac{n}{2}}. \tag{A.1}$$

Claim A.0.3. $x^{d(n,i,j)}$ *term is obtained only when: i)* $i_1 = i$*, ii)* $j_1 = i_2, \cdots, j_{n-1} = i_n$*, iii)* $j_n = j$.

Let us first rewrite $p_{\mathbf{C}}(x)$ as follows:

$$p_{\mathbf{C}}(x) = \sum_{\substack{i_1=1\cdots t,\cdots,i_n=1\cdots s \\ j_1=1\cdots s,\cdots,j_n=1\cdots t}} \mathbf{A}^{(1)}_{i_1,j_1}\mathbf{B}^{(1)}_{i_2,j_2}\cdots\mathbf{A}^{(n/2)}_{i_{n-1},j_{n-1}}\mathbf{B}^{(n/2)}_{i_n,j_n} \; x^{(s-1+j_1-i_2)+\cdots+i_1 s^{\frac{n}{2}}t^{\frac{n}{2}-1}+j_n s^{\frac{n}{2}}t^{\frac{n}{2}}}. \tag{A.2}$$

Note that we get the maximum degree when $i_1 = t-1, s-1+j_1-i_2 = 2s-2, \cdots, j_n = t-1$. Hence,

$$\begin{aligned}\text{max deg of } p_{\mathbf{C}}(x) &= 2s-2+s(2t-2)+\cdots+ \\ &\quad s^{n/2-1}t^{n/2-1}(2s-2)+(t-1)s^{n/2}t^{n/2-1} \\ &\quad +(t-1)s^{n/2}t^{n/2} \\ &= s^{n/2}t^{n/2-1}+s^{n/2}t^{n/2+1}-2 \\ &= k(n,s,t)-1.\end{aligned}$$

This shows Claim A.0.1. To show Claim A.0.2, note that:

$$\mathbf{C}_{i,j} = \sum_{j_1,j_2,\cdots,j_{n-1}} \mathbf{A}^{(1)}_{i,j_1}\mathbf{B}^{(1)}_{j_1,j_2}\mathbf{A}^{(2)}_{j_2,j_3}\mathbf{B}^{(2)}_{j_3,j_4}\cdots\mathbf{A}^{(n/2)}_{j_{n-2},j_{n-1}}\mathbf{B}^{(n/2)}_{j_{n-1},j}.$$

Among the terms in the sum in (A.2), $\mathbf{C}_{i,j}$ is the sum of terms that are from the i-th row of the first matrix $\mathbf{A}^{(1)}$ and the j-th column on the last matrix $\mathbf{B}^{(n/2)}$, and that have the second index and the first index of two adjacent matrices matching, e.g., $j_1 = i_2$ and $j_2 = i_3$. By setting these $i_1, \cdots, i_n, j_1, \cdots, j_n$ values, we obtain (A.1).

Lastly, we want to show Claim A.0.3. Let d be the degree of x in (A.2)

$$\begin{aligned}d &= (s-1+j_1-i_2)+\cdots+s^{\frac{n}{2}-1}t^{\frac{n}{2}-1}(s-1+j_{n-1}-i_n) \\ &\quad +i_1 s^{\frac{n}{2}}t^{\frac{n}{2}-1}+j_n s^{\frac{n}{2}}t^{\frac{n}{2}},\end{aligned} \tag{A.3}$$

which can be rewritten as:

$$d = d_0 + d_1\cdot s + d_2\cdot st + \cdots + d_{n-1}\cdot s^{\frac{n}{2}}t^{\frac{n}{2}-1} + d_n\cdot s^{\frac{n}{2}}t^{\frac{n}{2}}, \tag{A.4}$$

where

$$
\begin{aligned}
d_0 &= d \mod s \\
d_1 &= (d - d_0)/s \mod t \\
d_2 &= (d - d_0 - d_1 \cdot s)/st \mod s \\
&\vdots \\
d_n &= (d - d_0 - d_1 \cdot t - \cdots - d_{n-1} \cdot s^{n/2} t^{n/2-1})/s^{n/2} t^{n/2}.
\end{aligned}
$$

We can think of this representation as a mixed radix system $\mathcal{D}$ with $n+2$ digits, $(d_0, d_1, \cdots, d_{n+1})$, which has an alternating radix $(t, s, t, s, \cdots, t, s)$. By substituting $d_0 = t-1, d_1 = s-1, \cdots, d_{n+1} = s-1$, we can confirm that the biggest number we can represent with (A.4) is $s^{n+1}t^{n+1} - 1 > k(n,s,t) - 1$. Also, from its construction, any number between 0 and $s^{n+1}t^{n+1} - 1$ can be uniquely determined by the pair $(d_0, d_1, \cdots, d_{n+1})$ (for more explanation, see Theorem 1 in [35]). Hence, any $0 \leqslant d \leqslant k(n,s,t) - 1$ can be uniquely represented with $(d_0, d_1, \cdots, d_{n+1})$.

Now, we want to show that $d = d(n,i,j)$ only when $d_0 = s-1, d_1 = t-1, \cdots, d_{n-3} = t-1, d_{n-2} = s-1$ and $d_{n-1} = i, d_n = j$. It is easy to see that $d_0 = d(n,i,j) \mod s = s-1$, and similarly $d_1 = (d(n,i,j) - d_0) \mod t = t-1$ and so on. Since i_1 varies only from 0 to $t-1$,

$$
\begin{aligned}
d_{n-1} &= (i \cdot s^{\frac{n}{2}} t^{\frac{n}{2}-1} + j \cdot s^{\frac{n}{2}} t^{\frac{n}{2}})/s^{n/2} t^{n/2-1} \mod t \\
&= (i + jt) \mod t \\
&= i.
\end{aligned}
$$

Finally, $d_n = (j \cdot s^{\frac{n}{2}} t^{\frac{n}{2}})/s^{\frac{n}{2}} t^{\frac{n}{2}} = j$. As there is only one unique representation of any d with a tuple $(d_0, d_1, \cdots, d_n)$, by comparing (A.3) and (A.4), we can conclude that $j_1 = i_2, \cdots, j_{n-1} = i_n$, and $i_1 = i, j_n = j$. This completes the proof.

■

In this section, we present a coding technique for multiplying n matrices (*n-matrix multipli-*

cation), i.e., computing

$$\mathbf{C} = \mathbf{D}^{(1)}\mathbf{D}^{(2)}\cdots\mathbf{D}^{(n)}. \tag{A.5}$$

We state the problem formally in Section A.1 and then explain why this is different from multiplying two matrices. Then, in Section A.2, we provide a new code construction called n-*matrix codes* which applies MatDot codes and Polynomial codes in an alternating fashion. With this construction, we show that we can achieve recovery threshold of $\Theta(m^{\lceil n/2 \rceil})$ (see Theorem A.2.1) followed by a complexity analysis in Section A.3. After that, we propose a Generalized n-matrix codes in Section A.4 which allows for both horizontal and vertical partitioning of all the matrices being multiplied and again explore the trade-off between recovery threshold (see Theorem A.4.1 in Section A.4) and communication and computation complexity (Section A.5).

A.1 Problem Statement

We consider a generalization of the system model of Section 2.1.2.1 with a master node, P worker nodes, and a fusion node, to multiply more than two matrices. Here the goal is to compute the product $\mathbf{C} = \prod_{i=1}^{n} \mathbf{D}^{(i)}$ of $N \times N$ square matrices, $\mathbf{D}^{(1)}, \cdots, \mathbf{D}^{(n)}$. As we will treat the matrices $\mathbf{D}^{(i)}$ with odd and even indices differently, we will denote the $\mathbf{D}^{(i)}$'s with odd indices as $\mathbf{A}^{(\lceil i/2 \rceil)}$ and the $\mathbf{D}^{(i)}$'s with even indices as $\mathbf{B}^{(i/2)}$ for all $i \in \{1, \cdots, n\}$. Using this notation, $\mathbf{C}$ can be written as:

$$\mathbf{C} = \begin{cases} \prod_{i=1}^{\frac{n}{2}} \mathbf{A}^{(i)}\mathbf{B}^{(i)} & \text{if } n \text{ is even,} \\ \left(\prod_{i=1}^{\lfloor \frac{n}{2} \rfloor} \mathbf{A}^{(i)}\mathbf{B}^{(i)}\right) \mathbf{A}^{(\lceil \frac{n}{2} \rceil)} & \text{if } n \text{ is odd.} \end{cases} \tag{A.6}$$

In our model, each worker can receive at most nN^2/m symbols from the master node, where each symbol is an element of $\mathbb{F}$. Specifically, for each matrix $\mathbf{D}^{(i)}$, each worker receives N^2/m symbols which are $\mathbb{F}$-linear combinations of the entries of the matrix. Similar to Section 2.1.2.2, the computational complexities of the operations at master, worker and fusion nodes, in terms of the parameters N, P, m, are required to be strictly less than the computational complexity of

a sequential algorithm that computes the product. The goal is to perform this matrix product utilizing faulty or straggling workers with as low recovery threshold as possible. Again, in the following discussion, we will assume that $|\mathbb{F}| > P$.

A.2 Codes for n-matrix multiplication

Theorem A.2.1 (Recovery threshold for n-matrix codes)**.** *For the matrix multiplication problem specified in Section A.1 computed on the system defined in Definition 2.1.1, there exists a code with a recovery threshold of*

$$k(n,m) = \begin{cases} 2m^{n/2} - 1 & \text{if } n \text{ is even,} \\ (m+1)m^{\lfloor \frac{n}{2} \rfloor} - 1 & \text{if } n \text{ is odd.} \end{cases} \tag{A.7}$$

Discussion on applicability of n-matrix codes:

Before describing the code construction for n-matrix multiplication, we first discuss when n-matrix multiplication codes can be useful despite having a recovery threshold that grows exponentially with n. First, note that as n-matrix multiplication is a chain of $(n-1)$ matrix-matrix multiplications, one may think that we can apply the coding techniques developed in the previous sections to each pairwise matrix multiplication instead of developing a new coding technique for n-matrix multiplication. For example, let us consider computing $\mathbf{C} = \mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}^{(2)}$. A master node can first encode $\mathbf{A}^{(1)}$ and $\mathbf{B}^{(1)}$ using MatDot codes and distribute encoded matrices to all the worker nodes and the fusion node can decode $\mathbf{E} = \mathbf{A}^{(1)}\mathbf{B}^{(1)}$ from the output of successful worker nodes. Then we again encode $\mathbf{E}$ and $\mathbf{A}^{(2)}$ using MatDot code and distribute encoded matrices to the worker nodes. Finally, the fusion node can reconstruct $\mathbf{C}$ by decoding the outputs of successful worker nodes. As you can see from this example, simply applying MatDot codes on each matrix-matrix multiplication requires two rounds of communication after computing $\mathbf{E} = \mathbf{A}^{(1)}\mathbf{B}^{(1)}$ and $\mathbf{C} = \mathbf{E}\mathbf{A}^{(2)}$. For n-matrix multiplication, it requires $n-1$ rounds of communication. This can be inefficient in the systems when the communication cost increases with number of rounds of communication (e.g., due to large communication setup overheads).

What we propose in this section is a coded n-matrix multiplication strategy which requires only one round of communication. Our main result in Theorem A.2.1 shows that n-matrix codes need $\Theta(m^{\lceil n/2 \rceil})$ successful nodes to recover the computation result. On the other hand, successively applying MatDot codes requires $\Theta(m)$ nodes to successfully recover the final result, which is is in scaling sense smaller than $\Theta(m^{\lceil n/2 \rceil})$ for large n. This suggests that n-matrix codes avoid intermediate communications at the cost of larger recovery threshold. When communication start-up cost is the main source of delay, one should use n-matrix codes, and when number of computation nodes is limited, one should sequentially apply coding strategy for two-matrix multiplication such as MatDot or PolyDot codes.

Moreover, in many applications such as power-iteration-based methods, one often prefers to compute $\mathbf{A}^n\mathbf{x}^{(0)}$ (where $\mathbf{x}^{(0)} \in \mathbb{R}^n$ is an initial vector) instead of calculating $\mathbf{A}^n$ due to higher computational complexity. Our suggested coded multiple matrix-matrix multiplications can be employed in such applications simply by letting $\mathbf{D}^{(1)} = \mathbf{D}^{(2)} = \ldots = \mathbf{D}^{(n)} = \mathbf{A}$. Further details about this idea can be found in [45]. Therefore, redundancy overhead used in our scheme can be useful in such scenarios for two main reasons: (i) Saving communication cost; and (ii) Providing robustness against stragglers.

We will now begin with simple examples for even and odd n. The first example shows the example for even n, and present a construction for general n.

Example A.2.1 (Multiplying 4 matrices ($n = 4, m = 2, k = 7$))**.** *Here, we give an example of multiplying 4 matrices and show that a recovery threshold of 7 is achievable. For $i \in \{1, 2\}$, matrix $\mathbf{A}^{(i)}$ is split vertically into sub-matrices $\mathbf{A}_0^{(i)}, \mathbf{A}_1^{(i)}$ each of dimension $N \times \frac{N}{2}$ as follows: $\mathbf{A}^{(i)} = \left[\mathbf{A}_0^{(i)}\ \mathbf{A}_1^{(i)}\right]$, while, for $i \in \{1, 2\}$, matrix $\mathbf{B}^{(i)}$ is split horizontally into sub-matrices $\mathbf{B}_0^{(i)}, \mathbf{B}_1^{(i)}$ each of dimension $\frac{N}{2} \times N$ as follows:*

$$\mathbf{B}^{(i)} = \begin{bmatrix} \mathbf{B}_0^{(i)} \\ \mathbf{B}_1^{(i)} \end{bmatrix}. \tag{A.8}$$

Notice that the product $\mathbf{C} = \prod_{i=1}^{2} \mathbf{A}^{(i)}\mathbf{B}^{(i)}$ *can now be written as*

$$\begin{aligned}\prod_{i=1}^{2} \mathbf{A}^{(i)}\mathbf{B}^{(i)} &= \left(\mathbf{A}^{(1)}\mathbf{B}^{(1)}\right)\left(\mathbf{A}^{(2)}\mathbf{B}^{(2)}\right) \\ &= \left(\mathbf{A}_0^{(1)}\mathbf{B}_0^{(1)} + \mathbf{A}_1^{(1)}\mathbf{B}_1^{(1)}\right)\left(\mathbf{A}_0^{(2)}\mathbf{B}_0^{(2)} + \mathbf{A}_1^{(2)}\mathbf{B}_1^{(2)}\right). \end{aligned} \tag{A.9}$$

Now, we define the encoding polynomials $p_{\mathbf{A}^{(i)}}(x), p_{\mathbf{B}^{(i)}}(x)$, $i \in \{1, 2\}$ *as follows:*

$$\begin{aligned} p_{\mathbf{A}^{(1)}}(x) &= \mathbf{A}_0^{(1)} + \mathbf{A}_1^{(1)}x, \\ p_{\mathbf{B}^{(1)}}(x) &= \mathbf{B}_0^{(1)}x + \mathbf{B}_1^{(1)}, \\ p_{\mathbf{A}^{(2)}}(x) &= \mathbf{A}_0^{(2)} + \mathbf{A}_1^{(2)}x, \\ p_{\mathbf{B}^{(2)}}(x) &= \mathbf{B}_0^{(2)}x + \mathbf{B}_1^{(2)}. \end{aligned} \tag{A.10}$$

From (A.10), we have

$$\begin{aligned} p_{\mathbf{A}^{(1)}}(x)p_{\mathbf{B}^{(1)}}(x) &= \mathbf{A}_0^{(1)}\mathbf{B}_1^{(1)} + (\mathbf{A}_0^{(1)}\mathbf{B}_0^{(1)} + \mathbf{A}_1^{(1)}\mathbf{B}_1^{(1)})x \\ &\quad + \mathbf{A}_1^{(1)}\mathbf{B}_0^{(1)}x^2, \\ p_{\mathbf{A}^{(2)}}(x)p_{\mathbf{B}^{(2)}}(x) &= \mathbf{A}_0^{(2)}\mathbf{B}_1^{(2)} + (\mathbf{A}_0^{(2)}\mathbf{B}_0^{(2)} + \mathbf{A}_1^{(2)}\mathbf{B}_1^{(2)})x \\ &\quad + \mathbf{A}_1^{(2)}\mathbf{B}_0^{(2)}x^2. \end{aligned} \tag{A.11}$$

From (A.9) along with (A.11), we can observe the following:

(i) the coefficient of x *in* $p_{\mathbf{A}^{(1)}}(x)p_{\mathbf{B}^{(1)}}(x)$ *is* $\mathbf{A}_0^{(1)}\mathbf{B}_0^{(1)} + \mathbf{A}_1^{(1)}\mathbf{B}_1^{(1)} = \mathbf{A}^{(1)}\mathbf{B}^{(1)}$,

(ii) the coefficient of x^2 *in* $p_{\mathbf{A}^{(2)}}(x^2)p_{\mathbf{B}^{(2)}}(x^2)$ *is the product* $\mathbf{A}_0^{(2)}\mathbf{B}_0^{(2)} + \mathbf{A}_1^{(2)}\mathbf{B}_1^{(2)} = \mathbf{A}^{(2)}\mathbf{B}^{(2)}$, *and*

(iii) the coefficient of x^3 *in* $p_{\mathbf{A}^{(1)}}(x)p_{\mathbf{B}^{(1)}}(x)p_{\mathbf{A}^{(2)}}(x^2)p_{\mathbf{B}^{(2)}}(x^2)$ *is the product* $\prod_{i=1}^{2} \mathbf{A}^{(i)}\mathbf{B}^{(i)}$ *(our desired output).*

Let $x_1, \cdots, x_P$ *be distinct elements of* $\mathbb{F}$, *the master node sends* $p_{\mathbf{A}^{(i)}}(x_r^i)$ *and* $p_{\mathbf{B}^{(i)}}(x_r^i)$, *for all* $i \in \{1, 2\}$, *to the* r*-th worker node,* $r \in \{1, \cdots, P\}$, *and the* r*-th worker node performs the multiplication* $\prod_{i=1}^{2} p_{\mathbf{A}^{(i)}}(x_r^i)p_{\mathbf{B}^{(i)}}(x_r^i)$ *and sends the output to the fusion node.*

Let worker nodes $1, \cdots, 7$ be the first 7 worker nodes to send their computation outputs to the fusion node, then the fusion node receives the matrices $\prod_{i=1}^{2} p_{\mathbf{A}^{(i)}}(x_r^i) p_{\mathbf{B}^{(i)}}(x_r^i)$ for all $r \in \{1, \cdots, 7\}$. Since these 7 matrices can be seen as 7 evaluations of the matrix polynomial $\prod_{i=1}^{2} p_{\mathbf{A}^{(i)}}(x^i) p_{\mathbf{B}^{(i)}}(x^i)$ of degree 6 at 7 distinct evaluation points $x_1, \cdots, x_7$, the coefficients of the matrix polynomial $\prod_{i=1}^{2} p_{\mathbf{A}^{(i)}}(x^i) p_{\mathbf{B}^{(i)}}(x^i)$ can be obtained using polynomial interpolation. This includes the coefficient of x^3, i.e., $\prod_{i=1}^{2} \mathbf{A}^{(i)} \mathbf{B}^{(i)}$. ■

Now we show an example for odd n.

Example A.2.2 (Multiplying 3 matrices ($n = 3, m = 2, k = 5$))**.** *Here, we give an example of multiplying 3 matrices and show that a recovery threshold of 5 is achievable. In this example, we have three input matrices $\mathbf{A}^{(1)}$, $\mathbf{B}^{(1)}$, and $\mathbf{A}^{(2)}$, each of dimension $N \times N$ and need to compute the product $\mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}^{(2)}$. First, the three input matrices are split in the same way as in Example A.2.1. The product $\mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}^{(2)}$ can now be written as*

$$\mathbf{C} = \mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}^{(2)} = \begin{bmatrix} \mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}_0^{(2)} & \mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}_1^{(2)} \end{bmatrix}, \tag{A.12}$$

where $\mathbf{A}^{(1)}\mathbf{B}^{(1)} = \mathbf{A}_0^{(1)}\mathbf{B}_0^{(1)} + \mathbf{A}_1^{(1)}\mathbf{B}_1^{(1)}$.

Now, we define the encoding polynomials $p_{\mathbf{A}^{(1)}}(x), p_{\mathbf{B}^{(1)}}(x), p_{\mathbf{A}^{(2)}}(x)$ as follows:

$$\begin{aligned} p_{\mathbf{A}^{(1)}}(x) &= \mathbf{A}_0^{(1)} + \mathbf{A}_1^{(1)} x, \\ p_{\mathbf{B}^{(1)}}(x) &= \mathbf{B}_0^{(1)} x + \mathbf{B}_1^{(1)}, \\ p_{\mathbf{A}^{(2)}}(x) &= \mathbf{A}_0^{(2)} + \mathbf{A}_1^{(2)} x. \end{aligned} \tag{A.13}$$

From (A.13), we have

$$\begin{aligned} p_{\mathbf{A}^{(1)}}(x) p_{\mathbf{B}^{(1)}}(x) p_{\mathbf{A}^{(2)}}(x^2) &= \mathbf{A}_0^{(1)}\mathbf{B}_1^{(1)}\mathbf{A}_0^{(2)} \\ &+ (\mathbf{A}_0^{(1)}\mathbf{B}_0^{(1)} + \mathbf{A}_1^{(1)}\mathbf{B}_1^{(1)})\mathbf{A}_0^{(2)} x \\ &+ (\mathbf{A}_1^{(1)}\mathbf{B}_0^{(1)}\mathbf{A}_0^{(2)} + \mathbf{A}_0^{(1)}\mathbf{B}_1^{(1)}\mathbf{A}_1^{(2)}) x^2 \\ &+ (\mathbf{A}_0^{(1)}\mathbf{B}_0^{(1)} + \mathbf{A}_1^{(1)}\mathbf{B}_1^{(1)})\mathbf{A}_1^{(2)} x^3 + \mathbf{A}_1^{(1)}\mathbf{B}_0^{(1)}\mathbf{A}_1^{(2)} x^4. \end{aligned} \tag{A.14}$$

From (A.14), we can observe the following:

(i) the coefficient of x in $p_{\mathbf{A}^{(1)}}(x)p_{\mathbf{B}^{(1)}}(x)p_{\mathbf{A}^{(2)}}(x^2)$ is the product $\mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}_0^{(2)}$, and

(ii) the coefficient of x^3 in $p_{\mathbf{A}^{(1)}}(x)p_{\mathbf{B}^{(1)}}(x)p_{\mathbf{A}^{(2)}}(x^2)$ is the product $\mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}_1^{(2)}$.

From (A.12), these two coefficients suffice to recover $\mathbf{C}$. Let $x_1, \cdots, x_P$ be distinct elements of $\mathbb{F}$, the master node sends $p_{\mathbf{A}^{(i)}}(x_r^i)$, for all $i \in \{1, 2\}$, and $p_{\mathbf{B}_1}(x_r)$ to the r-th worker node, $r \in \{1, \cdots, P\}$, where the r-th worker node performs the multiplication $p_{\mathbf{A}^{(1)}}(x_r)p_{\mathbf{B}^{(1)}}(x_r)p_{\mathbf{A}^{(2)}}(x_r^2)$ and sends the output to the fusion node.

Let worker nodes $1, \cdots, 5$ be the first 5 worker nodes to send their computation outputs to the fusion node, then the fusion node receives the matrices $p_{\mathbf{A}^{(1)}}(x_r)p_{\mathbf{B}^{(1)}}(x_r)p_{\mathbf{A}^{(2)}}(x_r^2)$ for all $r \in \{1, \cdots, 5\}$. Since these 5 matrices can be seen as 5 evaluations of the polynomial $p_{\mathbf{A}^{(1)}}(x)p_{\mathbf{B}^{(1)}}(x)p_{\mathbf{A}^{(2)}}(x^2)$ of degree 4 at five distinct evaluation points $x_1, \cdots, x_5$, the coefficients of the matrix polynomial $p_{\mathbf{A}^{(1)}}(x_r)p_{\mathbf{B}^{(1)}}(x_r)p_{\mathbf{A}^{(2)}}(x_r^2)$ can be obtained using polynomial interpolation. This includes the coefficients of x and x^3, i.e., $\mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}_0^{(2)}$ and $\mathbf{A}^{(1)}\mathbf{B}^{(1)}\mathbf{A}_1^{(2)}$. ■

Next, we present a code construction for n-matrix multiplication for general n and m.

Construction A.2.1. *[n-matrix codes]*

***Splitting of input matrices**: for every $i \in \{1, \cdots, \lceil \frac{n}{2} \rceil\}$ and $j \in \{1, \cdots, \lfloor \frac{n}{2} \rfloor\}$, $\mathbf{A}_i$ and $\mathbf{B}_j$ are split as follows*

$$\mathbf{A}^{(i)} = \begin{bmatrix} \mathbf{A}_1^{(i)} & \mathbf{A}_2^{(i)} & \cdots & \mathbf{A}_m^{(i)} \end{bmatrix}, \quad \mathbf{B}^{(j)} = \begin{bmatrix} \mathbf{B}_1^{(j)} \\ \mathbf{B}_2^{(j)} \\ \vdots \\ \mathbf{B}_m^{(j)} \end{bmatrix}, \tag{A.15}$$

where, for $k \in \{1, \ldots, m\}$, $\mathbf{A}_k^{(i)}, \mathbf{B}_k^{(j)}$ are $N \times N/m$ and $N/m \times N$ dimensional matrices, respectively.

***Master node (encoding)**: Let $x_1, x_2, \ldots, x_{P-1}$ be arbitrary distinct elements of $\mathbb{F}$. For $i \in \{1, \cdots, \lceil \frac{n}{2} \rceil\}$, define $p_{\mathbf{A}^{(i)}}(x) = \sum_{j=1}^{m} \mathbf{A}_j^{(i)} x^{j-1}$, and, for $i \in \{1, \cdots, \lfloor \frac{n}{2} \rfloor\}$, define $p_{\mathbf{B}^{(i)}}(x) = \sum_{j=1}^{m} \mathbf{B}_j^{(i)} x^{m-j}$. For $r \in \{1, 2, \ldots, P\}$, the master node sends to the r-th worker the evaluations,*

$p_{\mathbf{A}^{(i)}}(x_r^{m^{i-1}})$ *and* $p_{\mathbf{B}^{(j)}}(x_r^{m^{j-1}})$*, for all* $i \in \{1, \cdots, \lceil \frac{n}{2} \rceil\}$ *and* $j \in \{1, \cdots, \lfloor \frac{n}{2} \rfloor\}$.

Worker nodes*: For* $i \in \{1, \cdots, \lceil \frac{n}{2} \rceil\}$*, define*

$$p_{\mathbf{C}^{(i)}}(x) = \begin{cases} p_{\mathbf{A}^{(i)}}(x) p_{\mathbf{B}^{(i)}}(x) & \text{if } i \in \{1, \cdots, \lfloor \frac{n}{2} \rfloor\}, \\ p_{\mathbf{A}^{(i)}}(x) & \text{if } n \text{ is odd and } i = \lceil \frac{n}{2} \rceil. \end{cases} \tag{A.16}$$

For $r \in \{1, 2, \ldots, P\}$*, the* r*-th worker node computes the matrix product* $\Pi_{i=1}^{\lceil \frac{n}{2} \rceil} p_{\mathbf{C}^{(i)}}(x_r^{m^{i-1}})$ *and sends it to the fusion node on successful completion.*

Fusion node (decoding)*: If* n *is even, the fusion node uses outputs of any* $2m^{\frac{n}{2}} - 1$ *successful workers to compute the coefficient of* $x^{m^{n/2}-1}$ *in the matrix polynomial* $\Pi_{i=1}^{\frac{n}{2}} p_{\mathbf{C}^{(i)}}(x^{m^{i-1}})$*, and if* n *is odd, the fusion node uses outputs of any* $m^{\lfloor \frac{n}{2} \rfloor}(m+1) - 1$ *successful workers to compute the coefficients of* $x^{jm^{\lfloor \frac{n}{2} \rfloor}-1}$*, for all* $j \in \{1, \cdots, m\}$*, in the matrix polynomial* $\Pi_{i=1}^{\lceil \frac{n}{2} \rceil} p_{\mathbf{C}^{(i)}}(x^{m^{i-1}})$ *(the feasibility of this step will be shown later in the proof of Theorem A.2.1).*

If the number of successful workers is smaller than $2m^{\frac{n}{2}} - 1$ *for even* n *or smaller than* $m^{\lfloor \frac{n}{2} \rfloor}(m+1) - 1$ *for odd* n*, the fusion node declares a failure.*

Remark A.2.1. The coefficient of $x^{m^i - m^{i-1}}$ in $p_{\mathbf{C}^{(i)}}(x^{m^{i-1}})$, for any $i \in \{1, \cdots, \lfloor \frac{n}{2} \rfloor\}$, is $\sum_{j=1}^{m} \mathbf{A}_j^{(i)} \mathbf{B}_j^{(i)} = \mathbf{A}^{(i)} \mathbf{B}^{(i)}$.

Remark A.2.2. A reader might wonder why there is a difference between odd-valued and even-valued n, and if one can be reduced to the other by introducing an identity matrix of dimensions $N \times N$ in the n-matrix multiplication problem. In this work, we have an assumption that the matrices being multiplied are not known in advance and may even be chosen by an adversary. If it is known in advance that one of the matrices is an identity matrix or even a matrix with a special structure, e.g., a Toeplitz matrix (essentially convolution), then alternative coding techniques might be applicable altogether, which we hope to explore as a future work. Here, we assume that none of the matrices are known to us, and we aim to find a general scheme. When $n = 2$, the n-matrix codes is exactly MatDot codes. When $n = 3$, (e.g., multiplying $\mathbf{ABC}$), it is Polynomial codes applied to $\mathbf{AB}$ and $\mathbf{C}$, followed by MatDot codes. It reduces to simply computing $\mathbf{AB}$ when we know that the third matrix $\mathbf{C}$ is identity, but without the hindsight, we

still have to encode the identity matrix, resulting in a bigger recovery threshold than multiplying two matrices.

A.3 Complexity Analyses of n-matrix codes (Construction A.2.1)

Encoding/decoding complexity: Decoding requires interpolating a $2m^{n/2} - 2$ degree polynomial if n is even or a $m^{\lfloor \frac{n}{2} \rfloor}(m+1) - 2$ degree polynomial if n is odd for each element in the matrix. Using polynomial interpolation algorithms of complexity $\mathcal{O}(k \log^2 k)$ [65], or $\mathcal{O}(k \log^2 k \log\log k)$ [63], where $k = k(n,m)$ as defined in (A.7), complexity per matrix element is $\mathcal{O}(m^{\lceil \frac{n}{2} \rceil} \log^2 m^{\lceil \frac{n}{2} \rceil}) \log\log m^{\lceil \frac{n}{2} \rceil})$. Thus, for N^2 elements, the decoding complexity is $\mathcal{O}(N^2 m^{\lceil \frac{n}{2} \rceil} \log^2 m^{\lceil \frac{n}{2} \rceil} \log\log m^{\lceil \frac{n}{2} \rceil})$.

Encoding for each worker requires performing n additions, each adding m scaled matrices of size N^2/m, for an overall encoding complexity for *each worker* of $\mathcal{O}(mnN^2/m) = \mathcal{O}(nN^2)$. Thus, the overall computational complexity of encoding for P workers is $\mathcal{O}(nN^2P)$.

Each worker's computational cost: Each worker multiplies n matrices of dimensions $N \times N/m$ and $N/m \times N$. For any worker r with $r \in \{1, \cdots, P\}$, the multiplication can be performed as follows:

Case 1**:** n is even

In this case, worker r wishes to compute the product:

$$p_{\mathbf{A}^{(1)}}(x_r) p_{\mathbf{B}^{(1)}}(x_r) p_{\mathbf{A}^{(2)}}(x_r^m) p_{\mathbf{B}^{(2)}}(x_r^m) \cdots$$
$$p_{\mathbf{A}^{(n/2)}}(x_r^{m^{n/2-1}}) p_{\mathbf{B}^{(n/2)}}(x_r^{m^{n/2-1}}).$$

Worker r does this multiplication in the following order:

1. Compute $p_{\mathbf{B}^{(i)}}(x_r^{m^{i-1}}) p_{\mathbf{A}^{(i+1)}}(x_r^{m^i})$ for all $i \in \{1, \cdots, n/2-1\}$ with a total complexity of $\mathcal{O}(nN^3/m^2)$.
2. Compute the product of the output matrices of the previous step with a total complexity of $\mathcal{O}(nN^3/m^3)$. Call this product matrix $\mathbf{D}$. Notice that $\mathbf{D}$ has a dimension of $N/m \times N/m$.

3. Compute $p_{\mathbf{A}^{(1)}}(x_r)\mathbf{D}$ with complexity $\mathcal{O}(N^3/m^2)$. Call this product matrix $\mathbf{E}$. Notice that $\mathbf{E}$ has a dimension of $N \times N/m$.

4. Compute $\mathbf{E}\, p_{\mathbf{B}^{(n/2)}}(x_r^{m^{n/2-1}})$ with complexity $\mathcal{O}(N^3/m)$.

Hence, the overall computational complexity per worker for even n is:

$$\mathcal{O}(\max(nN^3/m^2, nN^3/m^3, N^3/m^2, N^3/m)) = \mathcal{O}(\max(nN^3/m^2, N^3/m)).$$

Case 2**:** n is odd

In this case, worker r wishes to compute the product:

$$p_{\mathbf{A}^{(1)}}(x_r)p_{\mathbf{B}^{(1)}}(x_r)\cdots p_{\mathbf{A}^{((n-1)/2)}}(x_r^{m^{(n-3)/2}})$$
$$p_{\mathbf{B}^{((n-1)/2)}}(x_r^{m^{(n-3)/2}})p_{\mathbf{A}^{((n+1)/2)}}(x_r^{m^{(n-1)/2}}).$$

Worker r does this multiplication in the following order:

1. Compute $p_{\mathbf{B}^{(i)}}(x_r^{m^{i-1}})p_{\mathbf{A}^{(i+1)}}(x_r^{m^i})$ for all $i \in \{1, \cdots, (n-1)/2\}$ with a total complexity of $\mathcal{O}(nN^3/m^2)$.

2. Compute the product of the output matrices of the previous step with a total complexity of $\mathcal{O}(nN^3/m^3)$. Call this product matrix $\mathbf{D}$. Notice that $\mathbf{D}$ has a dimension of $N/m \times N/m$.

3. Compute $p_{\mathbf{A}^{(1)}}(x_r)\mathbf{D}$ with complexity $\mathcal{O}(N^3/m^2)$.

Hence, the overall computational complexity per worker for odd n is

$$\mathcal{O}(\max(nN^3/m^2, nN^3/m^3, N^3/m^2)) = \mathcal{O}(nN^3/m^2).$$

In conclusion, the computational complexity per worker is $\mathcal{O}(\max(nN^3/m^2, N^3/m))$ if n is even, and $\mathcal{O}(nN^3/m^2)$ if n is odd[1].

Communication cost: The master node communicates total of $\mathcal{O}(nPN^2/m)$ symbols to the worker nodes, and the fusion node receives $\mathcal{O}(m^{\lfloor \frac{n}{2} \rfloor}N^2)$ symbols from the successful worker nodes.

[1]The expressions for even n and odd n are different due to the last step in the even n case where we compute the matrix multiplication of dimension $N \times N/m$ and $N/m \times N$, which has computational complexity of $\mathcal{O}(N^3/m)$

A.4 Codes for Generalized n-matrix multiplication

Here, we give another code construction for n-matrix multiplication which is a generalization of the code construction given in the previous section. The new construction allows us to split input matrices more flexibly and trades off communication and computation (similar to PolyDot codes in Section 2.1.2.6 for two matrices). The results presented here are an improvement over [32], and are built on techniques from [26, 132].

Theorem A.4.1 (Recovery threshold for Generalized n-matrix codes)**.** *For the matrix multiplication problem specified in Section A.1 and computed on the system defined in Definition 2.1.1, there exists a code with a recovery threshold of*

$$k(n,s,t) = \begin{cases} s^{\frac{n}{2}}t^{\frac{n}{2}+1} + s^{\frac{n}{2}}t^{\frac{n}{2}-1} - 1 & \text{if } n \text{ is even,} \\ s^{\frac{n+1}{2}}t^{\frac{n+1}{2}} + s^{\frac{n-1}{2}}t^{\frac{n-1}{2}} - 1 & \text{if } n \text{ is odd} \end{cases} \tag{A.17}$$

for any integers s, t that satisfy $m = st$.

Proof. Here, we only derive the proof for the case of even n. The proof for odd n can be derived in a similar manner with minor differences in the expressions. What we have to show to complete the proof are as follows:

Claim A.4.2. *The maximum degree of $p_{\mathbf{C}}(x)$ is $s^{\frac{n}{2}}t^{\frac{n}{2}+1} + s^{\frac{n}{2}}t^{\frac{n}{2}-1} - 1$.*

Claim A.4.3. *$\mathbf{C}_{i,j}$ is the coefficient of $x^{d(n,i,j)}$ for $i, j = 1, \cdots, t$ where*

$$d(n,i,j) = s - 1 + s(t-1) + st(s-1) + \cdots + i \cdot s^{\frac{n}{2}}t^{\frac{n}{2}-1} + j \cdot s^{\frac{n}{2}}t^{\frac{n}{2}}. \tag{A.18}$$

Claim A.4.4. *$x^{d(n,i,j)}$ term is obtained only when: i) $i_1 = i$, ii) $j_1 = i_2, \cdots, j_{n-1} = i_n$, iii) $j_n = j$.*

Let us first rewrite $p_{\mathbf{C}}(x)$ as follows:

$$p_{\mathbf{C}}(x) = \sum_{\substack{i_1=1\cdots t,\cdots,i_n=1\cdots s \\ j_1=1\cdots s,\cdots,j_n=1\cdots t}} \mathbf{A}^{(1)}_{i_1,j_1}\mathbf{B}^{(1)}_{i_2,j_2}\cdots\mathbf{A}^{(n/2)}_{i_{n-1},j_{n-1}}\mathbf{B}^{(n/2)}_{i_n,j_n} \\ x^{(s-1+j_1-i_2)+\cdots+i_1 s^{\frac{n}{2}}t^{\frac{n}{2}-1}+j_n s^{\frac{n}{2}}t^{\frac{n}{2}}}. \tag{A.19}$$

Note that we get the maximum degree when $i_1 = t-1, s-1+j_1-i_2 = 2s-2, \cdots, j_n = t-1$. Hence,

$$\begin{aligned}\text{max deg of } p_{\mathbf{C}}(x) &= 2s-2+s(2t-2)+\cdots+ \\ &\quad s^{n/2-1}t^{n/2-1}(2s-2)+(t-1)s^{n/2}t^{n/2-1} \\ &\quad +(t-1)s^{n/2}t^{n/2} \\ &= s^{n/2}t^{n/2-1}+s^{n/2}t^{n/2+1}-2 \\ &= k(n,s,t)-1.\end{aligned}$$

This shows Claim A.4.2. To show Claim A.4.3, note that:

$$\mathbf{C}_{i,j} = \sum_{j_1,j_2,\cdots,j_{n-1}} \mathbf{A}^{(1)}_{i,j_1}\mathbf{B}^{(1)}_{j_1,j_2}\mathbf{A}^{(2)}_{j_2,j_3}\mathbf{B}^{(2)}_{j_3,j_4}\cdots\mathbf{A}^{(n/2)}_{j_{n-2},j_{n-1}}\mathbf{B}^{(n/2)}_{j_{n-1},j}.$$

Among the terms in the sum in (A.19), $\mathbf{C}_{i,j}$ is the sum of terms that are from the i-th row of the first matrix $\mathbf{A}^{(1)}$ and the j-th column on the last matrix $\mathbf{B}^{(n/2)}$, and that have the second index and the first index of two adjacent matrices matching, e.g., $j_1 = i_2$ and $j_2 = i_3$. By setting these $i_1,\cdots,i_n,j_1,\cdots,j_n$ values, we obtain (A.18).

Lastly, we want to show Claim A.4.4. Let d be the degree of x in (A.19)

$$\begin{aligned}d &= (s-1+j_1-i_2)+\cdots+s^{\frac{n}{2}-1}t^{\frac{n}{2}-1}(s-1+j_{n-1}-i_n) \\ &\quad +i_1 s^{\frac{n}{2}}t^{\frac{n}{2}-1}+j_n s^{\frac{n}{2}}t^{\frac{n}{2}},\end{aligned} \tag{A.20}$$

which can be rewritten as:

$$d = d_0 + d_1\cdot s + d_2\cdot st + \cdots + d_{n-1}\cdot s^{\frac{n}{2}}t^{\frac{n}{2}-1} + d_n\cdot s^{\frac{n}{2}}t^{\frac{n}{2}}, \tag{A.21}$$

where

$$
\begin{aligned}
d_0 &= d \mod s \\
d_1 &= (d - d_0)/s \mod t \\
d_2 &= (d - d_0 - d_1 \cdot s)/st \mod s \\
&\vdots \\
d_n &= (d - d_0 - d_1 \cdot t - \cdots - d_{n-1} \cdot s^{n/2} t^{n/2-1})/s^{n/2} t^{n/2}.
\end{aligned}
$$

We can think of this representation as a mixed radix system $\mathcal{D}$ with $n+2$ digits, $(d_0, d_1, \cdots, d_{n+1})$, which has an alternating radix $(t, s, t, s, \cdots, t, s)$. By substituting $d_0 = t-1, d_1 = s-1, \cdots, d_{n+1} = s-1$, we can confirm that the biggest number we can represent with (A.21) is $s^{n+1}t^{n+1} - 1 > k(n,s,t) - 1$. Also, from its construction, any number between 0 and $s^{n+1}t^{n+1} - 1$ can be uniquely determined by the pair $(d_0, d_1, \cdots, d_{n+1})$ (for more explanation, see Theorem 1 in [35]). Hence, any $0 \leqslant d \leqslant k(n,s,t) - 1$ can be uniquely represented with $(d_0, d_1, \cdots, d_{n+1})$.

Now, we want to show that $d = d(n,i,j)$ only when $d_0 = s-1, d_1 = t-1, \cdots, d_{n-3} = t-1, d_{n-2} = s-1$ and $d_{n-1} = i, d_n = j$. It is easy to see that $d_0 = d(n,i,j) \mod s = s-1$, and similarly $d_1 = (d(n,i,j) - d_0) \mod t = t-1$ and so on. Since i_1 varies only from 0 to $t-1$,

$$
\begin{aligned}
d_{n-1} &= (i \cdot s^{\frac{n}{2}} t^{\frac{n}{2}-1} + j \cdot s^{\frac{n}{2}} t^{\frac{n}{2}})/s^{n/2} t^{n/2-1} \mod t \\
&= (i + jt) \mod t \\
&= i.
\end{aligned}
$$

Finally, $d_n = (j \cdot s^{\frac{n}{2}} t^{\frac{n}{2}})/s^{\frac{n}{2}} t^{\frac{n}{2}} = j$. As there is only one unique representation of any d with a tuple $(d_0, d_1, \cdots, d_n)$, by comparing (A.20) and (A.21), we can conclude that $j_1 = i_2, \cdots, j_{n-1} = i_n$, and $i_1 = i, j_n = j$. This completes the proof. ■

Remark A.4.1. If we substitute $st = m$ in (A.17), we get:

$$k(n,s,t) = \begin{cases} m^{\frac{n}{2}}(t+1) - t & \text{if } n \text{ is even,} \\ m^{\frac{n-1}{2}}(m+t) - t & \text{if } n \text{ is odd} \end{cases} \tag{A.22}$$

By plugging in $s = m, t = 1$, we can see that $k(n,s,t) = 2m^{n/2} - 1$ for n even, and $k(n,s,t) = m^{\frac{n+1}{2}} + m^{\frac{n-1}{2}} - 1$ for n odd. This matches the recovery threshold given in (A.7).

We now give a construction of Generalized n-matrix codes.

Construction A.4.1 (Generalized n-matrix multiplication code)**.**

Splitting of input matrices*: We split $\mathbf{A}_i$'s and $\mathbf{B}_i$'s as follows:*

$$\mathbf{A}^{(i)} = \begin{bmatrix} \mathbf{A}^{(i)}_{0,0} & \cdots & \mathbf{A}^{(i)}_{0,s-1} \\ \vdots & \ddots & \vdots \\ \mathbf{A}^{(i)}_{t-1,0} & \cdots & \mathbf{A}^{(i)}_{t-1,s-1} \end{bmatrix},$$
$$\mathbf{B}^{(i)} = \begin{bmatrix} \mathbf{B}^{(i)}_{0,0} & \cdots & \mathbf{B}^{(i)}_{0,t-1} \\ \vdots & \ddots & \vdots \\ \mathbf{B}^{(i)}_{s-1,0} & \cdots & \mathbf{B}^{(i)}_{s-1,t-1} \end{bmatrix}, \tag{A.23}$$

where $\mathbf{A}^{(i)}_{j,k}$'s have dimension $N/t \times N/s$ and $\mathbf{B}^{(i)}_{j,k}$'s have dimension $N/s \times N/t$.

Master node (encoding)*: Define the encoding polynomials as*

$$p_{\mathbf{A}^{(1)}}(z_1, z_2) = \sum_{i=0}^{t-1}\sum_{j=0}^{s-1} \mathbf{A}^{(1)}_{i,j} z_1^i z_2^j,$$
$$p_{\mathbf{B}^{(1)}}(z_2, z_3) = \sum_{i=0}^{s-1}\sum_{j=0}^{t-1} \mathbf{B}^{(1)}_{i,j} z_2^{s-1-i} z_3^j,$$
$$\vdots,$$
$$p_{\mathbf{B}^{(n/2)}}(z_n, z_{n+1}) = \sum_{i=0}^{s-1}\sum_{j=0}^{t-1} \mathbf{B}^{(n/2)}_{i,j} z_n^{s-1-i} z_{n+1}^j.$$

for n even, and

$$p_{\mathbf{A}^{(1)}}(z_1, z_2) = \sum_{i=0}^{t-1}\sum_{j=0}^{s-1} \mathbf{A}^{(1)}_{i,j} z_1^i z_2^j,$$

$$\vdots,$$

$$p_{\mathbf{B}^{((n-1)/2)}}(z_{n-1}, z_n) = \sum_{i=0}^{s-1}\sum_{j=0}^{t-1} \mathbf{B}^{((n-1)/2)}_{i,j} z_{n-1}^{s-1-i} z_n^j,$$

$$p_{\mathbf{A}^{((n+1)/2)}}(z_n, z_{n+1}) = \sum_{i=0}^{t-1}\sum_{j=0}^{s-1} \mathbf{A}^{((n-1)/2)}_{i,j} z_n^{t-1-i} z_{n+1}^j,$$

for n odd.

The master node sends to the r-th worker evaluations of $p_{\mathbf{A}^{(i)}}$'s, and $p_{\mathbf{B}^{(i)}}$'s at

$$z_1 = x^{s^{n/2}t^{n/2-1}}, z_2 = x, z_3 = x^s, \cdots,$$

$$z_n = x^{s^{n/2-1}t^{n/2-1}}, z_{n+1} = x^{s^{n/2}t^{n/2}} \quad \text{for } n \text{ even}, \tag{A.24}$$

$$z_1 = x^{s^{(n-1)/2}t^{(n-1)/2}}, z_2 = x, x_3 = x^s, \cdots,$$

$$z_n = x^{s^{(n-1)/2}t^{(n-3)/2}}, z_{n+1} = x^{s^{(n-1)/2}t^{(n+1)/2}} \quad \text{for } n \text{ odd}. \tag{A.25}$$

where x_r's are all distinct for $r \in \{1, 2, \ldots, P\}$.

***Fusion node (decoding)**: The fusion node uses outputs of any $k(n, s, t)$ successful workers (given in (A.17)) to compute the coefficients of $p_{\mathbf{C}}(z)$. If the number of successful workers is smaller than $k(n, s, t)$, the fusion node declares a failure.*

Remark A.4.2. The two strategies for n-matrix multiplication proposed in this work can be understood better in our general PolyDot framework (see Table A.1). Essentially, they differ in the substitutions for the variables $z_1, \cdots, z_{n+1}$ to convert the polynomial in n variables into a polynomial in a single variable for the ease of interpolation. The main intuition behind the substitutions of (A.24) and (A.25) is that for z_1 and z_{n+1}, their powers grow from 0 to $t-1$ (or $s-1$), while all the other terms have powers growing from 0 to $2s-2$ (or $2t-2$). Hence, to minimize the maximum degree of the product polynomial, it is best to assign high powers of x to z_1 and z_{n+1}. An alternate substitution could also be to start with $z_1 = x$ and then continue

Table A.1: Comparison of different strategies for multiplying n matrices using different substitutions in the general PolyDot framework when n is even.

	n-matrix codes	Generalized n-matrix codes	Alternate Substitution
Substitution	$z_1 = z_2 = x, z_3 = z_4 = x^m, \cdots, z_{n-1} = x_n = x^{m^{n/2-1}}, z_{n+1} = x^{m^{n/2}}$	$z_1 = x^{s^{n/2}t^{n/2-1}}, z_2 = x, x_3 = x^s, \cdots, z_n = x^{s^{n/2-1}t^{n/2-1}}, z_{n+1} = x^{s^{n/2}t^{n/2}}$	$z_1 = x, z_2 = x^t, z_3 = x^{st}, \cdots, z_{n+1} = x^{s^{n/2}t^{n/2}} =$
Recovery Threshold	$2m^{n/2} - 1$	$s^{\frac{n}{2}}t^{\frac{n}{2}+1} + s^{\frac{n}{2}}t^{\frac{n}{2}-1} - 1$	$s^{\frac{n}{2}}t^{\frac{n}{2}+1} + s^{\frac{n}{2}}t^{\frac{n}{2}} - t$

substituting $z_2 = x^t$, $z_3 = x^{st}$, $z_4 = x^{st^2}$, ..., $z_{n+1} = s^{\lfloor \frac{n}{2} \rfloor}t^{\lceil \frac{n}{2} \rceil}$. The recovery threshold resulting due to this substitution is given by:

$$k(n,s,t) = \begin{cases} s^{\frac{n}{2}}t^{\frac{n}{2}+1} + s^{\frac{n}{2}}t^{\frac{n}{2}} - t & \text{if } n \text{ is even,} \\ s^{\frac{n+1}{2}}t^{\frac{n+1}{2}} + s^{\frac{n-1}{2}}t^{\frac{n+1}{2}} - t & \text{if } n \text{ is odd} \end{cases} \tag{A.26}$$

for any integers s, t that satisfy $m = st$. This is slightly higher than the recovery threshold obtained in Theorem A.4.1. Thus, for $n > 2$, we can improve the recovery threshold by delving deeper into the order of the substitution.

A.5 Complexity Analysis of Generalized n-matrix codes

Encoding/decoding complexity: Encoding communication cost is $\mathcal{O}(nN^2P)$ as in Section A.3. Decoding complexity is $\mathcal{O}(\frac{N^2}{t^2}k(n,s,t)\log^2 k(n,s,t)\log\log k(n,s,t))$ (even case)
or $\mathcal{O}(\frac{N^2}{ts}k(n,s,t)\log^2 k(n,s,t)\log\log k(n,s,t))$ (odd case).

Communication Complexity: The master node sends out $\mathcal{O}(nPN^2/m)$ encoded symbols

in the beginning. After the completion of computation, each node has to send $\mathcal{O}(N^2/t^2)$ symbols to the fusion node. Hence, total number of symbols the fusion node receives is $k(n,s,t)\cdot N^2/t^2$. Let us first consider the case when n is even. By substituting (A.17), we obtain $k(n,s,t)N^2/t^2 = \mathcal{O}(m^{n/2}/t)$. This is the same trade-off we observed using PolyDot codes for single matrix-matrix multiplication. For a fixed m, recovery threshold $k(n,s,t)$ grows linearly with t while communication cost is inversely related to t (See Fig.2.4). When n is odd, we do not see such trade-off. Recovery threshold is always $m^{(n-1)/2}(m+t)-t = \mathcal{O}(m^{(n+1)/2})$ regardless of the choice of t. Communication cost on the other hand is $k(n,s,t)N^2/t^2 = \mathcal{O}(m^{(n+1)/2}/t^2 + m^{(n-1)/2}/t)$ which decreases with growing t. For instance, if $t=1$, communication cost is $\mathcal{O}(m^{(n+1)/2})$, and when $t=m$, communication cost is $\mathcal{O}(m^{(n-3)/2})$. This suggests that when n is odd, it is always better to choose $t=m$ as m grows to infinity.

Each worker's computation cost: Using the similar technique shown in Section A.3, we can show that each worker's computation complexity is at most $\mathcal{O}(\max(nN^3/m^{1.5}, N^3/m))$ for any choice of s,t. If we compare the computation complexity for encoding/decoding and the computation complexity at each worker node, we can see that as long as $N > \mathcal{O}(m^{n/2-1.5}\log m)$, encoding/decoding computation overhead is amortized.

Remark A.5.1. Our result given here splits $\mathbf{A}^{(i)}$'s into $s\times t$ grid of blocks and $\mathbf{B}^{(i)}$'s into $t\times s$ grid of blocks. However, it is not necessary that all matrices have to be split in the same fashion. For instance, $\mathbf{A}^{(1)}$ can be divided into $t_1\times s_1$ grid and $\mathbf{B}^{(1)}$ can be divided into $s_1\times t_2$ grid, and so on. In this more general setting $\mathbf{A}^{(i)}$'s are split into $t_i\times s_i$ grid and $\mathbf{B}^{(i)}$'s are split into $s_i\times t_{i+1}$ grid. Let us denote $\mathbf{s}=[s_1,\cdots,s_{n/2}], \mathbf{t}=[t_1,\cdots,t_{n/2+1}]$. Then Theorem A.4.1 can be rewritten as follows.

$$k(n,\mathbf{s},\mathbf{t}) = \begin{cases} (t_{n/2+1}+1/t_1)\prod_{i=1}^{n/2} s_i t_i - 1 & \text{if } n \text{ even},\\ (t_1 s_{(n+1)/2}+1)\prod_{i=1}^{(n-1)/2} s_i t_i - 1 & \text{if } n \text{ odd}. \end{cases} \tag{A.27}$$

Remark A.5.2. In this work we assumed that all matrices have size $N\times N$ for simplicity.

However, this assumption is not necessary in the results presented here. When we have matrices with different dimensions to multiply, splitting each matrix in a different way would be more beneficial. For example, when we multiply matrices $\mathbf{A}, \mathbf{B}$ with dimensions $N \times N$ and $N \times 2$, we can divide $\mathbf{A}$ into $t \times s$ grid and divide $\mathbf{B}$ into $s \times 1$ grid.

Bibliography

[1] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Osdi*, volume 10, page 24, 2010. 1.1

[2] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 185–198, 2013. 1.2

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of . . . , 2006. 2.2, 2.2, 2.2

[4] Amotz Bar-Noy and Shlomo Kipnis. Broadcasting multiple messages in simultaneous send/receive systems. *Discrete Applied Mathematics*, 1994. 3.2.2.4

[5] Anne Benoit, Thomas Herault, Valentin Le Fèvre, and Yves Robert. Replication is more efficient than you think. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. ISBN 9781450362290. 1.2

[6] Ake Bjorck and Victor Pereyra. Solution of Vandermonde systems of equations. *Mathe-*

matics of Computation, 24(112):893–903, 1970. 4

[7] Richard E Blahut. *Algebraic codes for data transmission*. Cambridge university press, 2003. 3.2.2.4

[8] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. Algorithmic Based Fault Tolerance Applied to High Performance Computing. 2008. 1.2

[9] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Sally McKee, and Radu Rugina. Compiler-enhanced incremental checkpointing for openmp applications. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009. 1.2

[10] William G Brown, Joseph Tierney, and Reuben Wasserman. Improvement of electronic-computer reliability through the use of redundancy. *IRE Transactions on Electronic Computers*, (3):407–416, 1961. 1.1

[11] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems*, 1997. 3.2.1.5, 3.2.1.7, 3.2.2.4, 3.2.6, 3.2.7

[12] Viveck R Cadambe and Arya Mazumdar. Bounds on the size of locally recoverable codes. *IEEE transactions on information theory*, 61(11):5787–5794, 2015. 3.1.1.1

[13] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations: an International Journal*, 1(1):5–28, 2014. 1.2

[14] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13), 2007. 3.2.2.4

[15] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating

compute-intensive applications with GPUs and FPGAs. In *IEEE Symposium on Application Specific Processors*, pages 101–107, 2008. 2.2

[16] Zizhong Chen. Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. *ACM SIGPLAN Notices*, 48(8):167–176, 2013. ISSN 0362-1340. doi: 10.1145/2442516.2442533. 1.2

[17] Zizhong Chen and Jack Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, 2006. 1.2

[18] Y-H Choi and Miroslaw Malek. A fault-tolerant fft processor. *IEEE Transactions on Computers*, 37(5):617–621, 1988. 1.2

[19] Phillip Colella. Defining software requirements for scientific computing. 2004. 2.2

[20] William Dally. High-performance hardware for machine learning. *NIPS Tutorial*, 2015. 2.2

[21] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. pages 162–171, 2011. doi: 10.1145/1995896.1995923. 1.2

[22] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56 (2):74–80, 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408794. 1.1

[23] Prajjwal Devkota and AL Narasimha Reddy. Performance of quantized congestion notification in tcp incast scenarios of data centers. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 235–243. IEEE, 2010. 3

[24] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9): 4539–4551, 2010. 2.2

[25] Lara Dolecek, Frederic Sala, et al. Channel coding methods for non-volatile memories. *Foundations and Trendsextregistered in Communications and Information Theory*, 13(1): 1–128, 2016. 2.2

[26] 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 *IEEE International Symposium on Information Theory (ISIT)*, pages 1585–1589, 2018. 2.1.3, A.4

[27] Sanghamitra Dutta, Viveck Cadambe, and Pulkit Grover. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *Advances In Neural Information Processing Systems*, 2016. 2.2

[28] Sanghamitra Dutta, Ziqian Bai, Haewon Jeong, Tze Meng Low, and Pulkit Grover. A unified coded deep neural network training strategy based on generalized polydot codes for matrix multiplication. *arXiv preprint arXiv:1811.10751*, 2018. (document), 1.2, 2.2, 2.6

[29] Sanghamitra Dutta, Mohammad Fahim, Farzin Haddadpour, Haewon Jeong, Viveck Cadambe, and Pulkit Grover. On the optimal recovery threshold of coded matrix multiplication. *IEEE Transactions on Information Theory*, 2019. 1.2, 2.2, 3.1.1.1, 3.2.1.6, 4.1

[30] Peter Elias. Computation in the presence of noise. *IBM Journal of Research and Development*, 2(4):346–353, 1958. 1.2

[31] William S Evans and Leonard J Schulman. Signal propagation and noisy circuits. *IEEE Transactions on Information Theory*, 45(7):2367–2373, 1999. 2.2

[32] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover. On the optimal recovery threshold of coded matrix multiplication. In *IEEE Communication, Control, and Computing (Allerton)*, pages 1264–1270, Oct 2017. A.4

[33] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and

Ron Brightwell. rmpi: increasing fault resiliency in a message-passing environment. *Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488*, 2011. 1.2

[34] Kurt Ferreira, Jon Stearley, James H Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 44. ACM, 2011. 1.2

[35] A. S. Fraenkel. Systems of numeration. *The American Mathematical Monthly*, 92(2): 105–114, 1985. A, A.4

[36] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation". 3.2.2.2

[37] Javier Garcia-Frias and Wei Zhong. Approaching shannon performance by iterative decoding of linear codes with low-density generator matrix. *IEEE Communications Letters*, 7(6):266–268, 2003. 3

[38] Walter Gautschi. How (un) stable are vandermonde systems. *Asymptotic and computational analysis*, 124:193–210, 1990. 3.1.1

[39] Al Geist. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum*, 10:2–3, 2016. 1.1

[40] Israel Gohberg and Vadim Olshevsky. The fast generalized Parker-Traub algorithm for inversion of Vandermonde and related matrices. *Journal of Complexity*, 13(2):208–234, 1997. 4

[41] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 58(11):6925–6934, 2012. 3.1.1.1, 3.1.1.2

[42] Parikshit Gopalan, Cheng Huang, Bob Jenkins, and Sergey Yekhanin. Explicit maximally recoverable codes with locality. *IEEE Trans. Information Theory*, 60(9):5245–5256, 2014. 3.1.1.1

[43] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009. 2.2

[44] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017. 1.1

[45] Farzin Haddadpour, Yaoqing Yang, Viveck Cadambe, and Pulkit Grover. Cross-Iteration Coded Computing. In *IEEE Communication, Control, and Computing (Allerton)*, pages 196–203, 2018. A.2

[46] M. Haikin and R. Zamir. Analog coding of a source with erasures. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 2074–2078. IEEE, 2016. 3.1.1

[47] Doug Hakkarinen and Zizhong Chen. Algorithmic cholesky factorization fault recovery. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010. 1.2

[48] Wael Halbawi, Zihan Liu, and Babak Hassibi. Balanced reed-solomon codes for all parameters. In *2016 IEEE Information Theory Workshop (ITW)*, pages 409–413. IEEE, 2016. 3

[49] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Solving the straggler problem for iterative convergent parallel ml. 2015. 1.1

[50] Roger W Hockney. A fast direct solution of poisson's equation using Fourier analysis. *Journal of the ACM (JACM)*, 12(1), 1965. 2.2

[51] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage (TOS)*, 9(1):3, 2013. 3.1.1.1

[52] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984. ISSN 0018-9340. doi: 10.1109/tc.1984.1676475. (document), 1.2, 2.1.1, 2.1.1, 2.1, 2.2, 2.2, 3.2.1.4

[53] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R De Supinski, and Rudolf Eigenmann. Mcrengine: a scalable checkpointing system using data-aware aggregation and compression. *Scientific Programming*, 21(3-4):149–163, 2013. 1.2

[54] H. Jeong and P. Grover. Energy-adaptive codes. In *Annual Allerton Conference on Communication, Control, and Computing*, 2015. doi: 10.1109/ALLERTON.2015.7446995. 1.4.1

[55] H. Jeong and P. Grover. Energy-adaptive error correcting for dynamic and heterogeneous networks. *Proceedings of the IEEE*, 107(4):765–777, April 2019. ISSN 1558-2256. doi: 10.1109/JPROC.2019.2898366. 1.4.1

[56] H Jeong, TM Low, and P Grover. Masterless coded computing: A fully-distributed coded fft algorithm. *Communication, Control, and Computing (Allerton)*, 2018. 2.2

[57] Haewon Jeong, Christopher G Blake, and Pulkit Grover. Energy-adaptive polar codes: Trading off reliability and decoder circuit energy. In *Information Theory (ISIT), 2017 IEEE International Symposium on*, pages 2608–2612. IEEE, 2017. 1.4.1

[58] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al.

Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019. 1.1

[59] Jing-Yang Jou and Jacob A Abraham. Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures. *Proceedings of the IEEE*, 74(5), 1986. 2.2

[60] Jorn Justesen and Tom Hoholdt. Analysis of iterated hard decision decoding of product codes with reed-solomon component codes. In *2007 IEEE Information Theory Workshop*, pages 174–177. IEEE, 2007. 3.2.1

[61] C. Karakus, Y. Sun, S. Diggavi, and W. Yin. Straggler Mitigation in Distributed Optimization through Data Encoding. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5440–5448, 2017. 1.2

[62] I Kaufman. The inversion of the Vandermonde matrix and transformation to the Jordan canonical form. *IEEE Transactions on Automatic Control*, 14(6):774–777, 1969. 4

[63] Kiran S Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011. 2.1.2.5, 2.1.2.7, A.3

[64] Yongjune Kim, Abhishek A Sharma, Robert Mateescu, Seung-Hwan Song, Zvonimir Z Bandic, James A Bain, and BVK Vijaya Kumar. Locally rewritable codes for resistive memories. *IEEE Journal on Selected Areas in Communications*, 34(9):2470–2485, 2016. 2.2

[65] H. T. Kung. Fast evaluation and interpolation. Technical report, Carnegie Mellon University, 1973. 2.1.2.5, 2.1.2.7, A.3

[66] K. Lee, C. Suh, and K. Ramchandran. High-dimensional coded matrix multiplication. In *IEEE International Symposium on Information Theory (ISIT)*, pages 2418–2422, 2017. 2.2

[67] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran. Speeding up

distributed machine learning using codes. *IEEE Transactions on Information Theory*, 64 (3):1514–1529, 2018. 1.2, 2.1.1, 2.1.1, 2.2, 2.2, 3.1.2.1, 3.2.1.4, 3.2.1, 4.1

[68] Harvey Leff and Andrew F Rex. *Maxwell's Demon 2 Entropy, Classical and Quantum Information, Computing*. CRC Press, 2002. 1.1

[69] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr. Coded mapreduce. In *Communication, Control, and Computing (Allerton), 2015 53rd Annual Allerton Conference on*, pages 964–971. IEEE, 2015. 2.2

[70] Songze Li, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. A unified coding framework for distributed computing with straggling servers. In *Globecom Workshops (GC Wkshps), 2016 IEEE*. IEEE, 2016. 2.2

[71] Songze Li, Sucha Supittayapornpong, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. Coded terasort. *6th International Workshop on Parallel and Distributed Computing for Large Scale Machine Learning and Big Data Analytics*, 2017. (document), 2.2, 2.8

[72] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *arXiv preprint arXiv:1908.07873*, 2019. 1.1

[73] Fujitsu Limited. Fujitsu begins shipping supercomputer fugaku. press release, 2019. 1.1

[74] Shu Lin and Daniel J Costello. *Error control coding*, volume 2. Prentice hall, 2001. 1.3

[75] Shuangzhe Liu and Gõtz Trenkler. Hadamard, khatri-rao, kronecker and other matrix products. *Int. J. Inf. Syst. Sci*, 4(1):160–177, 2008. 3.1.2.4

[76] Michael Luby. Lt codes. In *null*, page 271. IEEE, 2002. 3

[77] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Elsevier, 1977. 3.1.2

[78] Michael Moldaschl, Karl E Prikopa, and Wilfried N Gansterer. Fault tolerant communication-optimal 2.5 d matrix multiplication. *Journal of Parallel and Distributed*

Computing, 104:179–190, 2017. 1.2

[79] Edward F Moore and Claude E Shannon. Reliable circuits using less reliable relays. *Journal of the Franklin Institute*, 262(3):191–208, 1956. 1.1

[80] Ihab Nahlus, Eric P Kim, Naresh R Shanbhag, and David Blaauw. Energy-efficient Dot-Product Computation using a Switched Analog Circuit Architecture. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 315–318, 2014. 2.2

[81] Frederique Oggier and Anwitaman Datta. Self-repairing homomorphic codes for distributed storage systems. In *INFOCOM, 2011 Proceedings IEEE*, pages 1215–1223. IEEE, 2011. 3.1.1.1

[82] Choong Gun Oh, Hee Yong Youn, and V. K. Raj. An efficient algorithm-based concurrent error detection for FFT networks. *IEEE Transactions on Computers*, 44(9), Sep 1995. ISSN 0018-9340. 2.2

[83] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999. 2.2

[84] D. S. Papailiopoulos and A. G. Dimakis. Locally repairable codes. *IEEE Transactions on Information Theory*, 60(10):5843–5855, Oct 2014. ISSN 0018-9448. doi: 10.1109/TIT.2014.2325570. 3.1.1.1

[85] FD Parker. Inverses of Vandermonde matrices. *The American Mathematical Monthly*, 71 (4):410–411, 1964. 4

[86] W Wesley Peterson and Michael O Rabin. On codes for checking logical operations. *IBM Journal of Research and Development*, 3(2):163–168, 1959. 1.2

[87] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G Andersen, Gregory R Ganger, Garth A Gibson, and Srinivasan Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST*, volume 8, pages 1–14, 2008. 3

[88] Stephen C Phillips, Vegard Engen, and Juri Papay. Snow white clouds and the seven

dwarfs. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 738–745, 2011. 2.2

[89] KV Rashmi, Nihar B Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 331–335. IEEE, 2013. 2.2

[90] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM SIGCOMM Computer Communication Review*, 44(4):331–342, 2015. 2.2

[91] Netanel Raviv, Rashish Tandon, Alex Dimakis, and Itzhak Tamo. Gradient coding from cyclic mds codes and expander graphs. In *International Conference on Machine Learning (ICML)*, pages 4302–4310, 2018. 1.2

[92] A. L. Narasimha Reddy and Prithviraj Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Transactions on Computers*, 39(10):1304–1308, 1990. 1.2

[93] Yongmao Ren, Yu Zhao, Pei Liu, Ke Dou, and Jun Li. A survey on tcp incast in data center networks. *International Journal of Communication Systems*, 27(8):1160–1172, 2014. 3

[94] Tom Richardson and Ruediger Urbanke. *Modern coding theory*. Cambridge university press, 2008. 1.3

[95] Ron Roth. *Introduction to coding theory*. Cambridge University Press, 2006. 1.3

[96] Robert D Ryne. On FFT-based convolutions and correlations, with application to solving poisson's equation in an open rectangular pipe. *arXiv preprint arXiv:1111.4971*, 2011. 2.2

[97] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003. 2.2

[98] Peter Sanders and Jop F Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Information Processing Letters*, 86(1), 2003. 3.2.2.4

[99] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes, editors, *High Performance Computing for Computational Science – VECPAR 2010*, pages 1–25, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19328-6. 3

[100] Claude Elwood Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949. 1.2

[101] Utsav Sheth, Sanghamitra Dutta, Malhar Chaudhari, Haewon Jeong, Yaoqing Yang, Jukka Kohonen, Teemu Roos, and Pulkit Grover. An Application of Storage-Optimal MatDot Codes for Coded Matrix Multiplication: Fast k-Nearest Neighbors Estimation. In *IEEE Big Data (Short Paper)*, 2018. 1.2, 2.1.2.5

[102] Natalia Silberstein, Ankit Singh Rawat, O Ozan Koyluoglu, and Sriram Vishwanath. Optimal locally repairable codes via rank-metric codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 1819–1823. IEEE, 2013. 3.1.1.1

[103] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011. 3.2.1.2

[104] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4): 354–356, 1969. 2.1.2.2, 5

[105] Leo Szilard. Über die entropieverminderung in einem thermodynamischen system bei eingriffen intelligenter wesen. *Zeitschrift für Physik*, 53(11-12):840–856, 1929. 1.1

[106] Itzhak Tamo and Alexander Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, 2014. 3.1.1.1, 3.1.1.2, 3.1.1, 3.1.1

[107] Itzhak Tamo, Alexander Barg, Sreechakra Goparaju, and Robert Calderbank. Cyclic lrc codes and their subfield subcodes. In *Information Theory (ISIT), 2015 IEEE International Symposium on*, pages 1262–1266. IEEE, 2015. 3.1.1.1

[108] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis. Gradient coding. In *Machine Learning Systems Workshop, Advances in Neural Information Processing Systems (NIPS)*, 2016. 1.2

[109] D. L. Tao and C. R. P. Hartmann. A novel concurrent error detection scheme for FFT networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(2), Feb 1993. ISSN 1045-9219. doi: 10.1109/71.207595. 2.2

[110] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005. 3.2.1.7, 3.2.1.7, 3.2.1.7

[111] Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected processor-node networks. *Journal of Parallel and Distributed Computing*, 68(7), 2008. 3.2.2.4, 3.2.2.4

[112] Joseph F Traub. Associated polynomials and uniform methods for the solution of linear problems. *Siam Review*, 8(3):277–301, 1966. 4

[113] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997. 3.2.1.2

[114] Vijay Vasudevan, Hiral Shah, Amar Phanishayee, Elie Krevat, David Andersen, Greg Ganger, and Garth Gibson. Solving tcp incast in cluster storage systems. In *The 7th USENIX Conference on File and Storage Technologies (FAST'09)*, 2009. 3

[115] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956. 1.1

[116] D. Wang, G. Joshi, and G. Wornell. Efficient Task Replication for Fast Response Times in

Parallel Computation. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):599–600, 2014. 2.1.2.5

[117] D. Wang, G. Joshi, and G. Wornell. Using Straggler Replication to Reduce Latency in Large-Scale Parallel Computing. *ACM SIGMETRICS Performance Evaluation Review*, 43(3):7–11, 2015. 1.2, 2.1.2.5

[118] Ning C Wang, Sujan K Gonugondla, Ihab Nahlus, Naresh R Shanbhag, and Eric Pop. Gdot: A graphene-based nanofunction for dot-product computation. In *VLSI Technology, 2016 IEEE Symposium on*, pages 1–2. IEEE, 2016. 2.2, 2.2

[119] Sinong Wang, Jiashang Liu, and Ness Shroff. Coded sparse matrix multiplication. *arXiv preprint arXiv:1802.03430*, 2018. 2.2

[120] Sying-Jyan Wang and N. K. Jha. Algorithm-based fault tolerance for FFT networks. *IEEE Transactions on Computers*, 43(7), Jul 1994. ISSN 0018-9340. 2.2

[121] Shmuel Winograd. Coding for logical operations. *IBM Journal of Research and Development*, 6(4):430–436, 1962. 1.2

[122] Shmuel Winograd and Jack D Cowan. *Reliable computation in the presence of noise*. Number 22. Mit Press Cambridge, Mass., 1963. (document), 1.1, 1.2, 1.1

[123] Lihao Xu, Vasken Bohossian, Jehoshua Bruck, and David G Wagner. Low-density mds codes and factors of complete graphs. *IEEE Transactions on Information theory*, 45(6): 1817–1826, 1999. 3

[124] Yaoqing Yang, Pulkit Grover, and Soummya Kar. Computing linear transformations with unreliable components. *IEEE Transactions on Information Theory*, 63(6), 2017. 2.2

[125] Yaoqing Yang, Pulkit Grover, and Soummya Kar. Coded distributed computing for inverse problems. In *Advances in Neural Information Processing Systems (NIPS)*, pages 709–719, 2017. 1.2, 3.1.1

[126] Yaoqing Yang, Malhar Chaudhari, Pulkit Grover, and Soummya Kar. Coded iterative

computing using substitute decoding. *arXiv preprint arXiv:1805.06046*, 2018. 2.2

[127] Yaoqing Yang, Pulkit Grover, and Soummya Kar. Coding for a single sparse inverse problem. In *2018 IEEE International Symposium on Information Theory (ISIT)*, pages 1575–1579. IEEE, 2018. 2.2

[128] Erlin Yao, Jiutian Zhang, Mingyu Chen, Guangming Tan, and Ninghui Sun. Detection of soft errors in LU decomposition with partial pivoting using algorithm-based fault tolerance. *The International Journal of High Performance Computing Applications*, 29(4): 422–436, 2015. ISSN 1094-3420. doi: 10.1177/1094342015578487. 1.2

[129] Min Ye and Emmanuel Abbe. Communication-computation efficient gradient coding. In *International Conference on Machine Learning (ICML)*, pages 5606–5615, 2018. 1.2

[130] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr. Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication. In *Advances In Neural Information Processing Systems (NIPS)*, pages 4403–4413, 2017. (document), 2.1.1, 2.2, 2.1.1, 2.1.2, 2.1.2.6, 2.2, 3.1.1.1, 3.1.2.1, 3.1.2.3, 4.1

[131] Qian Yu, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. Coded Fourier transform. *arXiv preprint arXiv:1710.06471*, 2017. 2.2

[132] Qian Yu, Mohammad Ali Maddah-Ali, and A Salman Avestimehr. Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding. In *IEEE International Symposium on Information Theory (ISIT)*, pages 2022–2026, 2018. 1.2, 1.4, 2.1.3, 2.2, A.4

[133] Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6. IEEE, 2012. 1.2