Hierarchical Coded Matrix Multiplication

Shahrzad Kiani, Nuwan Ferdinand, and Stark C. Draper

Abstract—In distributed computing systems slow working nodes, known as stragglers, can greatly extend finishing times. Coded computing is a technique that enables straggler-resistant computation. Most coded computing techniques presented to date provide robustness by ensuring that the time to finish depends only on a set of the fastest nodes. However, while stragglers do compute less work than non-stragglers, in real-world commercial cloud computing systems (e.g., Amazon’s Elastic Compute Cloud (EC2)) the distinction is often a soft one. In this paper, we develop hierarchical coded computing that exploits the work completed by all nodes, both fast and slow, automatically integrating the potential contribution of each. We first present a conceptual framework to represent the division of work amongst nodes in coded matrix multiplication as a cuboid partitioning problem. This framework allows us to unify existing methods and motivates new techniques. We then develop three methods of hierarchical coded computing that we term bit-interleaved coded computation (BICC), multilevel coded computation (MLCC), and hybrid hierarchical coded computation (HHICC). In this paradigm, each worker is tasked with completing a sequence (a hierarchy) of ordered subtasks. The sequence of subtasks, and the complexity of each, is designed so that partial work completed by stragglers can be used in, rather than ignored. We note that our methods can be used in conjunction with any coded computing method. We illustrate this showing how we can use our methods to accelerate all previously developed coded computing technique by enabling them to exploit stragglers. Under a widely studied statistical model of completion times, our approach realizes a 66% improvement in expected finishing time. On Amazon EC2, the gain was 28% when stragglers are simulated.

I. INTRODUCTION

The advent of large scale machine learning algorithms and data analytics has increased the demand for computation. Such computation often cannot be performed in a single computer due to limited processing power and available memory. Parallelization is necessary. In an idealized distributed setting highly parallelizable tasks can be accelerated proportional to the number of working nodes. However, in many cloud-based systems, slow working nodes, known as stragglers, present a bottleneck that can prevent the realization of faster compute times [5]. Recent studies show that for certain linear algebraic tasks, such as matrix multiplication, the effect of stragglers can be minimized through the use of error correction codes [6, 7, 8, 9, 10, 11]. This idea, termed coded computing, introduces redundant computations so that the completion of any fixed-cardinality subset of tasks suffices to realize the desired solution. Coded computing can greatly accelerate many machine learning algorithms such as those that involve computing gradients, thus accelerating the training of large-scale machine learning applications [12, 13, 14, 15, 16]. While matrix multiplication and gradient descent are two types of coded computing problems that have been studied, others include coded convolution [17], coded approximate computing [18], sparse coded matrix multiplication [19], and heterogeneous coded computing [20].

In most of the coded computing work to date a type of erasure model is assumed. Workers either complete their job or complete no work. While this can be a good model for hardware failure, it is not a perfect model for all decentralized computing systems. In many systems one observes a spectrum of completion speeds. One aim of this paper is to advocate for a more nuanced view of stragglers. A processing node may be slower than the average node, but yet may complete some work. On the other hand, it may be faster than the average node, a leader, and correspondingly able to complete more work. We illustrate one method for maximizing the contributions of both fast and slow workers in the context of accelerating matrix-matrix multiplications. We believe the general idea can be applied quite widely. In the remainder of this section, we discuss the prior work we build on, some contemporary work that thinks about how to exploit stragglers, and then we present our contributions.

A. Background: Stragglers and coded computing

A simple approach to deal with stragglers is to replicate tasks. This is equivalent to repetition coding. However, if the job can be linearly decomposed, the opportunity arises to introduce redundancy more efficiently through the use of error-correction codes. In particular, in [6] maximum distance separable (MDS) codes are used to develop a straggler-resistant method of vector-matrix multiplication. This idea is extended to matrix-matrix multiplication using product codes in [8]. In [2], coded computation based on polynomial interpolation is introduced. Polynomial codes outperform product codes in terms of their recovery threshold. This is the number of workers that are required to complete their tasks for the master to be able to recover the desired calculation. With memory per worker fixed, the recovery threshold of polynomial codes is further improved by MatDot codes [10], albeit at the cost of increased per-worker computation. In addition to MatDot coding, [10] introduces polyDot coding as a generalization and unification of polynomial and MatDot codes. PolyDot codes provide a tradeoff between the recovery threshold and the computation load assigned to each worker. In works subsequent to polyDot coding, [11] and [13] simultaneously arrived at new coding methods that can improve the recovery threshold of polyDot codes. These coding methods are respectively referred to as entangled polynomial and Generalized polyDot codes.

Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada. Emails: shahrzad.kianidehkordi@mail.utoronto.ca, \{nuwan.ferdinand and stark.draper\}@utoronto.ca.

This research was supported in part by a Discovery Research Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) and by an NSERC postdoctoral fellowship. It was presented in part in ISIT’18 [1], CWIT’19 [2], and ICML’19 [3].
As already mentioned, a drawback to many of the initial coded computing designs is that they rely only on the work completed by a set of the fastest workers. They ignore completely the work done by the slower workers. In the terminology of error correction coding, these slower nodes are modeled as erasures. However, in systems such as the Amazon Elastic Compute Cloud (EC2), we observe partial stragglers. Partial stragglers are slower, only able to complete partial tasks by the time at which the faster workers have completed their entire tasks. That said, the amount of work stragglers can complete may be non-negligible. Thus, it can be wasteful to ignore.

Recent literature (including our work) \cite{12, 1, 21, 2, 3, 4, 22, 23, 24} proposes methods to exploit the work completed by stragglers, rather than ignoring it. The underlying idea is to assign each worker a sequence of multiple small subtasks rather than a single large task. The master is able to complete the job by utilizing the computations completed by all workers, stragglers simply contribute less. The concept of exploiting partial straggler was first studied in \cite{12} such that each worker is assigned two groups of subtasks: naive and coded subtasks. Non-straggler workers process both naive and coded subtasks, while partial stragglers only complete naive tasks. In \cite{1, 21} each worker is tasked by a fully-coded series of subtasks, where all tasks are coded with respect to a single code. In \cite{1, 21} the vector matrix multiplication was first broken into computationally homogeneous subtasks; each subtask was then encoded using a specific code: an MDS code in \cite{1} and a rateless fountain code in \cite{21}. In our paper \cite{2} we further leverage the sequential computing nature of each worker by introducing the concept of hierarchical coding. This concept was explained based on the vector matrix multiplication problem using MDS code. In hierarchical coding, workers are all tasked to complete a hierarchy of coded subtasks, each subtask is coded separately. We extended special cases of hierarchical coding in \cite{3, 4}. More recent works that aim to exploit stragglers including \cite{22, 23, 24, 22} complement the idea presented in \cite{21} and the idea we present in \cite{1}. In \cite{22} each worker is tasked by a specified fraction of coded and uncoded computations. In \cite{23} multiple coded subtasks assigned to each worker are generated according to the characteristics of universally decodable matrices. In \cite{24} each worker is tasked with completing a fully-uncoded series of subtasks with respect to a predesigned computation order.

### B. Contribution

In this paper we develop three novel approaches to coded computing that can leverage the work completed by the average node, can exploit the work completed by the stragglers, and can allow fast worker to contribute even more to the overall computation. In analogy with some similarly-structured approaches in coded modulation we term these bit-interleaved coded computing (BICC), multilevel coded computing (MLCC), and hybrid hierarchical coded computing (HHCC) \footnote{BICC and MLCC were presented in part at \cite{1, 2, 3} and \cite{4}.}. As in all these approaches each processing unit is presented with a hierarchy of tasks, we use the unifying term hierarchical coding to refer to the methods in aggregate. Two ideas underlie our designs: assignment of multiple subtasks to each worker, and recognition of the inherent sequential processing nature of the workers. Before presenting hierarchical coding later in the paper, in Sec. III we first establish an equivalence between task allocation in coded matrix multiplication and a geometric problem of partitioning a rectangular cuboid into subcuboids. The allocation of tasks in the various coded matrix multiplication approaches developed to date correspond to different partitionings of the cuboid. The cuboid visualization will facilitate our extension to hierarchical coding.

We now sketch the observations that lead to our code designs. Our first observation is that by assigning each worker multiple smaller subtasks we can realize a more continuous completion process. This enables the exploitation of a broader spectrum of workers, including stragglers, and motivates the design of BICC. Our second observation is we recognize the inherently sequential processing nature of compute nodes. Once a worker is assigned multiple tasks, and due to sequential processing, more workers will finish their first assigned subtask than their second (or third, fourth, etc.). We exploit this observation in MLCC by grouping subtasks into levels, one subtask per level. Workers all start with their first level subtask. Since the most workers complete their first level subtask, we assign the first level the highest rate; later levels generally receive lower rates. So, while in BICC all subtasks are jointly encoded, in MLCC encoding distinct levels is independently. On the one hand, this means that BICC enjoys a more flexible recovery ability than MLCC, which result in a lower computation time. On the other hand, BICC suffers from more complicated decoding and higher communication overhead than MLCC. These characteristics roughly parallel bit-interleaved coded modulation (BICM) \cite{25} and multilevel coding (MLC) \cite{26}, hence the choice of names. We also introduce randomized MLCC (RMLCC) in which each worker randomly picks (without replacement) a level of computation and completes the encoded subtask pertinent to that level. Due to this random allocation mechanism and for large enough number of workers, the same number of workers on average are assigned with different permutation of orders. This mitigates the recovery ability of MLCC. To combine the strengths of BICC and MLCC we introduce a hybrid version. This version realizes an achievable design trade-off between computation, decoding, and communication times. We note that our methods can be used in conjunction with any coded computing method. We illustrate this showing how we can use our methods to accelerate all previously developed coded computing technique by enabling them to exploit stragglers. We prove both theoretically and experimentally that our proposed hierarchical techniques improve the total finishing time when applies to standard coded computing schemes. We numerically show that our method realizes a 60% improvement in the expected finishing time for a widely studied modeled of shifted exponential completion time of each subtask. On Amazon EC2, the gain was 28% when stragglers are simulated.
C. Outline

Other than a brief discussion of notation (next), the rest of the paper is laid out as follows. In Sec. III we describe a system model that is considered in this paper and discuss the relevant performance measures. In Sec. III we introduce our cuboid partitioning visualization and detail the state-of-the-art coded matrix multiplication approaches through this visualization. In Sec. IV we first introduce BICC, MLCC, and HHCC in detail and then compare these hierarchical coding schemes. In Sec. V we theoretically analyze the finishing time of non-hierarchical and hierarchical designs and provide a parameter design for BICC and MLCC. In Sec. VI we experimentally show that hierarchical coding decreases the total finishing time when compared to non-hierarchical coding. Finally, we conclude our contribution in Sec. VII.

D. Notation

Sets are denoted using calligraphic font, e.g., $S$. The cardinality of a finite set $S$ is denoted $|S|$. We use bold upper case, e.g., $A$, for matrices. The entry in the $i$th row and $j$th column of $A$, is denoted as $a_{i,j}$. The submatrix of $A$ is obtained by selecting out a collection of rows, $S_i$, and columns, $S_c$, is denoted $A_{S_i,S_c}$ where $S_i = S_i^\times S_i^c$. To denote the $i$th element of $S_i$ we write $s_{i,j}$, which is a row-index into the A matrix. Similarly, $S_{i,j}$ is column-index into a matrix. We also use the notation $[q] + c = \{1 + c, \ldots, q + c\}$ for the shifted index set, where $q,c \in \mathbb{Z}^+$; $[q]$ means $c = 0$.

II. SYSTEM MODEL

We consider the problem of multiplying two matrices $A \in \mathbb{R}^{N_x \times N_x}$ and $B \in \mathbb{R}^{N_y \times N_y}$ in a distributed coded system that consists of a central node, referred as the master, and $N$ individual nodes, called workers. We parallelize the computation of the matrix product $AB \in \mathbb{R}^{N_x \times N_y}$ among $N$ workers by providing each a subset of the data and requesting each to carry out specific computations. In Sec. II-A we detail our model of a distributed coded matrix multiplication system. In Sec. II-B we define the performance metrics we use to compare designs.

A. Distributed coded matrix multiplication

In the following we first explain the system model of a distributed coded matrix multiplication problem through six phases: data partitioning, encoding, distribution, worker computation, result aggregation, and decoding phases. This system model is summarized in Fig. 1. We then define the terminology we use and detail phases of two state-of-the-art coding schemes, polynomial [9] and MatDot [10] codes.

Phase 1 (Data partitioning): We first partition the overall computation of the matrix product $AB$ into $K$ equal-sized\footnote{The load allocation is assumed to be homogeneous; the extension of this assumption to a heterogeneous system is discussed in Sec. V-B} computations. The master will be able to recover the $AB$ product if and only if all $K$ computations are completed. To accomplish this the master first partitions the data by dividing $A$ and (separately) $B$ into, respectively, $M_x M_z$ and $M_y M_z$ equal-sized matrices of dimension $N_x/M_x \times N_x/M_x$ and $N_y/M_y \times N_y/M_y$:

\[
\begin{align*}
\{ A_{S_{ix} \times S_{ix}} \mid i \in [M_z]\},
\{ B_{S_{ij} \times S_{ik}} \mid j \in [M_z], k \in [M_y]\},
\end{align*}
\]

where $K = M_z M_x M_y, S_{ix} \times S_{ix}$, and $S_{ij} \times S_{ik}$ are respectively the subsets of (generally, but not necessarily) consecutive elements of $[N_x]$, $[N_x]$, and $[N_y]$. To ensure that the submatrices $A_{S_{ix} \times S_{ix}}$ and $B_{S_{ij} \times S_{ik}}$ partition the entire matrix $A$ (and $B$), these subsets must satisfy $\bigcup_{i \in [M_z], j \in [M_z], k \in [M_y]} S_{ix} \times S_{ij} \times S_{ik} = [N_x] \times [N_x] \times [N_y]$. Note that, in general the submatrices $A_{S_{ix} \times S_{ix}}$ and $B_{S_{ij} \times S_{ik}}$, $i \in [N_x], j \in [N_x], k \in [N_y]$, can have overlap; that simply would mean certain elements of the matrix $AB$ would be computed more than once. For conceptual clarity for the moment, we assume that they are disjoint.

This partitioning process can be reversed using a concatenation. For example, let’s assume that the matrix $A_{S_{ix} \times S_{ix}}$ contains all elements $a_{S_{ix} \times S_{ix}}$, where $i \in [N_x/M_x] + (i - 1)[N_x/M_x]$ and $i \in [N_x/M_x] + (i - 1)[N_x/M_x]$. Likewise, the matrix $B_{S_{ij} \times S_{ik}}$ contains all elements $b_{S_{ij} \times S_{ik}}$, where $j \in [N_x/M_y] + (j - 1)[N_x/M_y]$ and $j \in [N_x/M_y] + (j - 1)[N_x/M_y]$. We can therefore reverse the $A$ and $B$ partitioning by concatenating the matrices:

\[
\begin{align*}
A & = \begin{bmatrix}
A_{S_{i1} \times S_{x1}} & \cdots & A_{S_{i1} \times S_{xMz}} \\
\vdots & \ddots & \vdots \\
A_{S_{iMz} \times S_{x1}} & \cdots & A_{S_{iMz} \times S_{xMz}}
\end{bmatrix}, \\
B & = \begin{bmatrix}
B_{S_{x1} \times y1} & \cdots & B_{S_{x1} \times yM_y} \\
\vdots & \ddots & \vdots \\
B_{S_{xMz} \times y1} & \cdots & B_{S_{xMz} \times yM_y}
\end{bmatrix}. \tag{2}
\end{align*}
\]

Once the data $A$ and $B$ are partitioned, $K$ pairs of matrices can be matched up to yield the $AB$ product:

\[
\{ A_{S_{ix} \times S_{ix}} B_{S_{ij} \times S_{ik}} \mid i \in [M_z], j \in [M_z], k \in [M_y]\}.
\]

In particular, we now present two particular partitionings of the $AB$ product, which we refer to as inner-product and outer-product partitioning.

In inner-product partitioning the $A$ and $B$ matrices are respectively partitioned horizontally and vertically. That is $M_z = 1$. The $AB$ product is thereby divided into $K$ inner products, each between a group of rows in $A$ and a group of columns in $B$. One way to achieve $K$ inner products is to divide $A$ and $B$ into $\sqrt{K}$ equal-sized matrices\footnote{Assume $\sqrt{K} \approx |K|$. We ignore the integer effects to clarify the conceptual framework. In our implementation results we address integer effects.}.

We partition $A$ horizontally into $M_x = \sqrt{K}$ matrices $A_{S_{ix} \times [N_x]}$, where $i \in [\sqrt{K}]$ and $S_{xi} = [[N_x/\sqrt{K}]] + (i - 1)[N_x/\sqrt{K}]$. We partition $B$ vertically into $M_y = \sqrt{K}$ matrices $B_{[N_x] \times S_{iy}}$, where $i \in [\sqrt{K}]$ and $S_{iy} = [[N_x/\sqrt{K}]] + (i - 1)[N_x/\sqrt{K}]$. The $AB$ product is thereby divided into $K$ computations of matrix products $A_{S_{ix} \times [N_x]} B_{[N_x] \times S_{iy}}$, $i, j \in [\sqrt{K}]$.

In contrast to inner-product partitioning, in outer-product partitioning $A$ and $B$ are, respectively, divided vertically and horizontally into $M_z = K$ sub-matrices. That means that $A$ and $B$ are divided horizontally and vertically into $K$ equal-sized matrices $A_{S_{ix} \times [N_x]}$ and $B_{[N_x] \times S_{iy}}$, $i, j \in [\sqrt{K}]$.
Typically, each coding scheme corresponds to a specific data partitioning encoding phase of polynomial \[9\] and MatDot \[10\] codes.

Another option is to combine the inner-product and the outer-product ideas, partitioning \(A\) and \(B\) both horizontally and vertically, i.e., \(M_x, M_z, \) and \(M_y > 1\). We term this combinatorial data partitioning.

**Phase 2 (Encoding):** In this phase redundancy is introduced into computations so that the desired \(AB\) product can be recovered from a subset of completed tasks. To introduce such redundancy, the master encodes the partitioned data from (1) to generate \(N\) pairs of encoded matrices:

\[
\{ (\hat{A}(n), \hat{B}(n)) \mid n \in [N] \}.
\]

Typically, each coding scheme corresponds to a specific data partitioning structure. In Examples 1 and 2 we detail the encoding phase of polynomial \[9\] and MatDot \[10\] codes.

**Phase 3 (Distribution):** In this phase the master distributes the encoded data, sending \(\hat{A}(n)\) and \(\hat{B}(n)\) to the \(n\)th worker.

**Phase 4 (Worker computation):** Worker \(n\) computes the product \(\hat{A}(n)\hat{B}(n)\), and transmits the result to the master as soon as it is completed.

**Phase 5 (Results aggregation):** The master aggregates completed results from workers till it receives \(R\) out of \(N\) completed matrix products \(\hat{A}(n)\hat{B}(n)\). We note that \(R\) is a function of \(M_x, M_z, \) and \(M_y\). For example, in polyDot \[10\], entangled polynomial \[11\], and product \[8\] codes

\[
\begin{align*}
R_{\text{Poly}} &= M_x M_y, \\
R_{\text{Ent}} &= M_x M_z M_y + M_z - 1, \\
R_{\text{Product}} &= (M_x + M_y - 2)\sqrt{N} - (M_x - 1)(M_y - 1) + 1.
\end{align*}
\]

Polynomial codes \[9\] and MatDot codes \[10\] are special codes of polyDot codes, as \(M_z = 1\) in polynomial codes and \(M_x = M_y = 1\) in MatDot codes. The recovery thresholds of these two coding schemes therefore are

\[
\begin{align*}
R_{\text{Poly}} &= M_x M_y, \\
R_{\text{Mat}} &= 2M_z - 1.
\end{align*}
\]

**Phase 6 (Decoding):** The master recovers the \(AB\) product by implementing the decoding algorithm corresponding to the coding scheme used in the encoding phase. In Examples 1 and 2 we detail the decoding phase of polynomial \[9\] and MatDot \[10\] codes.

We now highlight the terminology we use to describe the system model of standard coded computing.

**Definition 1: (Information dimension):** We use \(K\) to denote the information dimension of a code. The information dimension refers to the number of useful (non-redundant) computations into which the main computational job is partitioned. In other words, completion of the computational job depends on the completion of at least \(K\) computations.

**Definition 2: (Block length):** The block length of a code refers to the total number of coded computational tasks that are encoded from \(K\) computations. In standard coded computation we use \(N\) to denote number of workers which also refers to the block length of a code.

Note that in this terminology, \(\frac{K}{N}\) refers as a rate of a code. That is, for \(K\) useful computations, the encoder generates \(N\) encoded tasks.

**Definition 3: (Recovery threshold):** The recovery threshold refers to the minimum number of encoded tasks that are required to be completed, so that the main computational job can be recovered. We use \(R\) to denote the recovery threshold of a code. (N.B. \(R\) is reserved for recovery threshold in coded computing, and not the rate.)

We note that in the existing literature often only two of the above parameters are specified, e.g., \(N\) and \(R\) (and not \(K\)). This is because in many of those works there is a functional relationship between \(K, N,\) and \(R\), e.g., \(K = R_{\text{Poly}}\) in polynomial codes \[9\]. In the present paper we introduce all three, \(K, N,\) and \(R\), so that our discussion can encompass all previous coding schemes. We now detail the system model of two specific coded matrix multiplication problems.

**Example 1 (polynomial coded matrix multiplication \[9\]):** To use polynomial coding, the master uses inner-product data (B) is divided into \(K\) matrices \(A_{[N_x] \times S_{zi}}\) \((B_{S_{zi} \times [N_y]}\), where \(i \in [K]\) and \(S_{zi} = [[N_z/K]] + (i - 1)[N_z/K]\). Therefore, \(M_x = M_y = 1\). In outer-product partitioning the \(AB\) product is divided into a set of \(K\) outer products, each between a group of columns in \(A\) and a group of rows in \(B\), i.e., \(A_{[N_x] \times S_{zi}} B_{S_{zi} \times [N_y]}\), where \(i \in [K]\).

Fig. 1: A master-worker model for distributed coded matrix multiplication problem.
partitioning. To allocate tasks to $N$ workers, the master applies polynomial codes (separately) to $A_{S_{s_i} \times [N_z]}$ and $B_{[N_z] \times S_y}$, where $i, j \in [\sqrt{K}]$. This generates $N$ encoded submatrices $\hat{A}(n)$ and $\hat{B}(n)$, where $n \in [N]$.

For example, if $K = 4$ ($\sqrt{K} = 2$),

$$
\hat{A}(n) = (A_{S_{s_1} \times [N_z]} + A_{S_{s_2} \times [N_z]}) n \in \mathbb{R}^{N_z \times N_z},
$$

$$
\hat{B}(n) = (B_{[N_z] \times S_{s_1}} + B_{[N_z] \times S_{s_2}} n^2 \in \mathbb{R}^{N_z \times N_z}.
$$

For polynomial codes of information dimension $K$, the recovery threshold is $R_{\text{poly}} = K$. For example, if $K = 4$, the master can decode from any $R_{\text{poly}} = 4$ encoded products.

For $K = 4$, the polynomial $\hat{A}(x)\hat{B}(x)$ is a polynomial of degree 3 and therefore can be recovered via polynomial interpolation from any 4 distinct values. Once the master receives any $R_{\text{poly}}$ results, it decodes the $AB$ matrix via polynomial interpolation [9].

**Example 2 (MatDot coded matrix multiplication [10]):** In contrast to polynomial codes, MatDot codes use outer-product data partitioning. The master leverages polynomials to generate product data partitioning. The master leverages polynomials to generate $N$ pairs of encoded submatrices ($\hat{A}(n), \hat{B}(n)$), where $n \in [N]$ from the $2K$ submatrices $A_{[N_z] \times S_{s_i}}$ and $B_{S_{s_i} \times [N_y]}$, where $i \in K$. For $K = 2$, the encoded submatrices are

$$
\hat{A}(n) = (A_{[N_z] \times S_{s_1}} + A_{[N_z] \times S_{s_2}} n \in \mathbb{R}^{N_z \times N_z},
$$

$$
\hat{B}(n) = (B_{S_{s_1} \times [N_y]} + B_{S_{s_2} \times [N_y]} n \in \mathbb{R}^{N_z \times N_y}.
$$

For MatDot codes, the recovery threshold is $R_{\text{Mat}} = 2K - 1$. For example, if $K = 4$, then $R_{\text{Mat}} = 7$. Once the master receives any $R_{\text{Mat}}$ results, it interpolates the polynomial $\hat{A}(n)\hat{B}(n)$ and recovers the $AB$ product.

**B. Performance measures**

To compare the performance of different coding schemes, we now introduce the measures we use. We start by grouping the measures into four categories: communication, worker computation, encoding, and decoding costs.

**Communications cost** We define the input communication load of the $n$th worker as the number of real numbers that the master distributes to the $n$th worker in the distribution phase. We use output communication load to describe the communication load in the results aggregation phase which is generally not equal to that of the distribution phase. We use $C_{\text{in}}$ and $C_{\text{out}}$ to denote the respective loads.

As an illustration, Table I summarizes $C_{\text{in}}$ and $C_{\text{out}}$ for the polynomial and MatDot coding examples of Sec. II-A.

| Example 1 | $C_{\text{in}}^{\text{enc}}$ | $C_{\text{in}}^{\text{comp}}$ | $C_{\text{out}}$ |
|-----------|-------------------------------|-------------------------------|-----------------|
| Example 2 | $\frac{N_z N_y}{K} + \frac{N_z N_y}{K}$ | $\frac{N_z N_y}{K}$ | $N_z N_y$ |

**TABLE I: Comparison of polynomial codes and MatDot codes across load measurements.**

communication time, the time to communicate a scalar real number between master and worker $n$ (in either direction). We assume a linear relation, $C_{\text{in}}^{\text{comm}}T_{\text{comm}}$, to compute the $n$th worker’s communication time for conveying $C_{\text{in}}^{\text{comm}}$ real numbers. In other words, given $T_{\text{comm}}$ it takes the master $C_{\text{in}}^{\text{comm}}T_{\text{comm}}$ sec to convey the input data consisting of $C_{\text{in}}^{\text{comm}}$ real numbers to the $n$th worker. Similarly, it takes the $n$th worker $C_{\text{out}}^{\text{comm}}T_{\text{comm}}$ sec to convey the output data $C_{\text{out}}$ to the master.

**Worker computation cost** Similar to communication cost, we define both load and latency measurements for worker computation cost. The computation load, denoted by $C_{\text{comp}}$, measures the number of multiply-and-accumulate operations that worker $n$ performs in the computation phase. For a fixed $K$, in both examples of Sec. II-A, the $n$th worker has the same computation load: $C_{\text{comp}} = N_x N_z N_y / K$, c.f. Table I.

The latency measurement indicates the amount of time it would take worker $n$ to compute a multiply-and-accumulate operation. We call this the computation time of the $n$th worker and denote it $T_{\text{comp}}$. We assume workers make linear progress. Given $T_{\text{comp}}$, worker $n$ would take $C_{\text{comp}}T_{\text{comp}}$ sec to perform $C_{\text{comp}}$ multiply-and-accumulate operations.

**Encoding cost** The encoding cost of a scheme counts the number of multiply-and-accumulate operations required. We use $C_{\text{enc}}$ to denote the encoding cost of worker $n$. In the polynomial code example, to generate $\hat{A}(n)$ we first multiply size $\left\lceil \frac{N_z}{\sqrt{K}} \right\rceil \times N_z$ matrices by a scalar. We then sum $\sqrt{K}$ matrices of dimensions $\left\lceil \frac{N_z}{\sqrt{K}} \right\rceil \times N_z$. This requires $N_z N_z$ multiply-and-accumulate operations. Similarly, to generate $\hat{B}(n)$, $N_z N_y$ multiply-and-accumulate operations are required. Therefore, the encoding cost of worker $n$ in the example is $C_{\text{enc}} = N_x N_z + N_z N_y$. This is also equal to $C_{\text{enc}}$ for the MatDot codes example. We note that, in the case of dealing with very large matrices, where $K \ll \min(N_x, N_z, N_y)$, the encoding cost is negligible in comparison to the computation load. In particular, $N_z N_z + N_z N_y \ll N_x N_z N_y / K$ in the two examples. Therefore, we ignore the encoding time when calculating the finishing time in Sec. III.

**Decoding cost** Decoding complexity depends on many factors such as the hardware specification and the decoding algorithm implementation. For instance, in both Examples 1 and 2 decoding complexity is governed by the complexity of interpolating a degree $R - 1$ polynomial, which is order $O((R - 1) \log^3(R - 1))$ [9].

**III. A unified geometric model**

We now present a conceptual framework wherein the decomposition of a matrix multiplication task into smaller computations can geometrically be visualized as the partitioning of a three-dimensional cuboid. This visualization would be used in the data partitioning phase of Sec. II-A. In this section we show
that the data partitioning phase in various prior coded matrix multiplication approaches, e.g., polynomial \cite{9}, MatDot \cite{10} codes, and others, corresponds to different partitions of the cuboid. Starting from this geometric perspective, in Sec. \[IV\] we design hierarchical coding in such a way that this idea can immediately be combined with all coded matrix multiplication schemes.

A. 3D visualization: Standard matrix multiplication

Standard (non-coded) matrix multiplication techniques for computing the product $AB$, where $A \in \mathbb{R}^{N_x \times N_z}$ and $B \in \mathbb{R}^{N_z \times N_y}$, require $N_x N_z N_y$ basic operations, each of which is a multiply-and-accumulate. This basic operation $g : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ is defined pointwise as $g(a, b, c) = ab + c$. One method to compute each entry of $AB$ is iteratively to apply the basic operation $N_z$ times to calculate an inner product (cf., Alg. \[I\]).

\begin{algorithm}
\caption{\([C] = \text{MatMult}(A, B)\)}
1 \textbf{Input:} $A \in \mathbb{R}^{N_x \times N_z}$, $B \in \mathbb{R}^{N_z \times N_y}$, \\
2 \textbf{Output:} $C \in \mathbb{R}^{N_x \times N_y}$, \\
\hspace{1cm} 1: \textbf{for all} $i_x \in [N_x], i_y \in [N_y]$; \\
\hspace{1.5cm} 2: \hspace{0.5cm} c_{i_x,i_y} = 0$ \\
\hspace{1.5cm} 3: \textbf{for} $i_z \in [N_z]$; \\
\hspace{2.5cm} 4: \hspace{0.5cm} c_{i_x,i_y} = g(a_{i_x,i_z}, b_{i_z,i_y}, c_{i_x,i_y})$ \\
\hspace{1cm} 5: \textbf{end for} \\
\hspace{1cm} 6: \textbf{end forall}
\end{algorithm}

Each basic operation is indexed by a positive integer triple $(i_x, i_z, i_y) \in \mathcal{I} = [N_x] \times [N_z] \times [N_y]$ such that the pairs $(i_x, i_z)$ and $(i_z, i_y)$ index the entries of $A$ and $B$ that serve as the $a$ and $b$ entries in $g(a, b, c)$. In 3D space each integer triple $(i_x, i_z, i_y)$ can be visualized as indexing a unit cube situated within a rectangular cuboid of integer edge lengths $(N_x, N_z, N_y)$ (cf., Fig. \[23\]).

B. 3D visualization: Data partitioning in coded computing

We now employ the 3D cuboid visualization to understand the data partitioning phase of coded matrix multiplication. The partitioning of the $AB$ product into $K = M_x M_z M_y$ computations can be visualized as a partitioning of the 3D cuboid. This partitioning \textit{slices} the cuboid into $K$ equal-sized subcuboids by making $M_x - 1$ parallel cuts along the $x$-axis, $M_z - 1$ parallel cuts along the $z$-axis, and $M_y - 1$ parallel cuts along the $y$-axis. There are eight possible cuboid partitions. Each corresponds to a distinct approach to coded matrix multiplication. Each of the eight partitionings is defined by slicing the 3D cuboid along a specific subset of directions $\{x, z, y\}$. All the eight partitionings are represented by a tree in Fig. \[3\]. Slicing or not slicing along each axis is illustrated, respectively, by labeling edge of the tree with a “1” or a “0”.

For instance, product codes \cite{8} and polynomial codes \cite{9} correspond to slicing along the $x$- and $y$-axes. This results in inner-product partitioning. The decomposition used in the data partitioning phase of polynomial coding example, discussed in Sec. \[II\] slices the 3D cuboid into $K$ equal-sized subcuboids by making $\sqrt{K} - 1$ parallel cuts along the $x$-axis and $\sqrt{K} - 1$ parallel cuts along the $y$-axis. This partitioning is depicted in Fig. \[25\] for $K = 4$. MatDot codes \cite{10} slice along the $z$-axis, which defines the outer-product partitioning. As is illustrated in Fig. \[25\] the 3D cuboid partitioning for MatDot codes involves $K - 1 = 3$ slices along the $z$-axis.

PolyDot codes \cite{10} and entangled polynomial codes \cite{11} slice the 3D cuboid along all $(x, y, z)$ axes. This is what we termed combinatorial partitioning (cf., Fig. \[24\]). Slicing along either $(x,z)$- or $(y,z)$- axes are special cases of slicing along the $(x, y, z)$ axes, where only one matrix is split both horizontally and vertically and the other matrix is split in only one direction. Slicing along $x$- or $y$-axis specifically models coded vector-matrix or coded matrix-vector multiplication. It also, more generally, models coded matrix multiplication in which only one matrix is split. The last category, denoted by empty set, represents an $(N, N)$ repetition code in which each of the $N$ workers is tasked with completing the entire $AB$ product.

The $K$ equal-sized subcuboids into which the 3D cuboid is partitioned correspond to $K$ distinct computations. We use the term \textit{information block} to refer to each of these subcuboids. Note that the number of information blocks is equal to the information dimension $K$ of the error correction code that we introduced earlier.

IV. HIERARCHICAL CODED MATRIX MULTIPLICATION

Building from our cuboid perspective, we now introduce hierarchical coded computing. The intuition underlying hierarchical coding is to subdivide the single task that each worker is assigned with in (non-hierarchical) coded computing into a number of smaller subtasks. The division into a number of subtasks allows each worker to make progress through its assigned tasks in a more continuous, incremental manner. Since each worker progresses through a sequence or \textit{hierarchy} of subtasks, we term these hierarchical methods. Once we break each (original) task into the smaller subtasks we have flexibility in how to apply coding across the subtasks. We present three alternatives: BICC (bit-interleaved coded computing), MLCC (multilevel coded computing), and HHCC (hybrid hierarchical coded computing). Each has its respective strengths.

In Fig. \[4\] we illustrate a toy example for each scheme. Figure \[4a\] corresponds to BICC, Fig. \[4b\] to MLCC, and Fig. \[4c\] to HHCC. We assume that we have $N = 3$ workers, and we need two third of computations to be completed in order to complete the main computational job. In all three figures each worker is provided $P = 4$ encoded subtasks. In traditional non-hierarchical coded computing $P = 1$, and there would be a single task for each worker. As Fig. \[5a\] illustrates, one hierarchical approach is BICC that applies a single code across all the subtasks, such that the block length of the code, which equals the number of encoded subtasks, is $NP = 12$. Two
third of these subtasks, which results in the recovery threshold $R_{BICC} = 8$, are required to be completed.

A second hierarchical approach is MLCC that groups the subtasks into $L = 4$ levels of computation. Each level contains a single subtask for each worker, hence $L = P$ (cf. Fig. 4(b)). In MLCC the master collects the required eight completed subtasks through four smaller codes each of which has a block length $N = 3$. The total recovery threshold is split into per-level recovery thresholds $R_l$ such that $\sum_l R_l = 8$. We note that since all workers start with their first (encoded) subtask and sequentially work their way through their $P$ subtasks, more workers will complete their first subtask than their $P$-th. Therefore, we (in general) use a higher-rate code, i.e., a larger recover threshold $R_l$, for $l = 1$ than for $l = P$.

Finally, as is illustrated in Fig. 4(c) in HHCC we combine these two ideas. In this example, there are $L_{HCCC} = 2$ levels of computation, but each worker also has $P_i = 2$ subtasks per level, $l \in [2]$. The block length of both levels is thus $P_i N = 6$, while it is 12 in BICC and 3 in MLCC. The master collects eight required subtasks through two codes with recovery thresholds $R_1 = 5$ and $R_2 = 3$, so that $\sum_l R_l = 8$.

The categorization of traditional non-hierarchical coded computing, BICC, MLCC, and HHCC is presented in Table 11. There are two design axes. The first is the number of codes used. A single code, which corresponds to a single level of the hierarchy (non-hierarchical and BICC) or multiple codes, i.e., multiple levels (MLCC and HHCC). The other design axis is the number of encoded subtasks each worker is assigned per level. Either a single subtask (non-hierarchical and MLCC) or multiple subtasks (BICC and HHCC) per level. In Table 11 we summarize the notation we use for both hierarchical and non-hierarchical schemes.

### A. Bit-interleaved coded matrix multiplication

In the following we detail the six phases of the system model for BICC with parameters $(L_{BICC} = 1, K_{BICC}, P, N, R_{BICC})$.

**Data partitioning phase and cuboid visualization:** Similar to non-hierarchical coding, in BICC $L_{BICC} = 1$. The master directly decomposes the single-level computational job into $K_{BICC}$ equal-sized sub-computations. In contrast to the information dimension $K$ in non-hierarchical coding, in BICC the information dimension is $K_{BICC} = PK$. This is equal to the number of equal-sized sub-computations into which the matrix product $AB$ is partitioned. From the cuboid perspective, to realize a $K_{BICC}$ we first assume that $K_{BICC} = M_x BICC M_z BICC M_y BICC$. Based on this assumption, we partition the cuboid into $K_{BICC}$ equal-sized information blocks with $M_x BICC - 1$ slices along the $x$-axis and $M_z BICC - 1$ slices along the $z$- and $y$-axes, respectively.

Such a partitioning is equivalent to partitioning the matrix $A$ into $M_x BICC M_z BICC$ equal-sized matrices $A_{S_{x,i} \times S_{z,j}}$, where $i \in [M_x BICC]$ and $j \in [M_z BICC]$. Likewise, the master partitions the matrix $B$ into $M_z BICC M_y BICC$ equal-sized matrices $B_{S_{z,j} \times S_{y,k}}$, where $j \in [M_z BICC]$ and $k \in [M_y BICC]$. The matrix $A_{S_{x,i} \times S_{z,j}}$ is of dimension $N_x/M_x BICC \times N_z/M_z BICC$, and $B_{S_{z,j} \times S_{y,k}}$ is of dimension $N_z/M_z BICC \times N_y/M_y BICC$.

As before, we comment that these information blocks can be allowed to overlap. For simplicity of presentation, we assume they are disjoint.
That same worker next computes \( \hat{y} \) yielding different performance in terms of the input and output instance, if we follow [9], polynomials are used to generate distinct points. We use \( P \) for all \( n \). Note that given \( \hat{N} \) distinct workers, i.e., no single matrix is given to two workers.

**Encoding phase:** The master encodes \( 2K_{\text{BICC}} \) matrices \( \hat{A}_{S_k} \) and (separately) \( \hat{B}_{S_k}, \) \( k \in [K_{\text{BICC}}] \), to generate \( PN \) pairs of encoded matrices \( (\hat{A}(n), \hat{B}(n)), \) where \( n \in [PN] \). For instance, if we follow [9], polynomials are used to generate encoded matrices. To generate encoded matrices for worker \( n \in [N] \), the master evaluates polynomials \( A(x) \) and \( B(x) \) at \( P \) distinct points. We use \( x \in \{(n-1)P+1, \ldots, (n-1)P+P\} \).

**Distribution phase:** The master transmits \( P \) distinct pairs of encoded matrices \( (\hat{A}((n-1)P+p), \hat{B}((n-1)P+p)), \) where \( p \in [P] \) to worker \( n \in [N] \). All matrices are distributed to distinct workers, i.e., no single matrix is given to two workers. We note that \( \hat{A}(n) \) and \( \hat{B}(n) \) are, respectively, of dimensions \( N_x/M_{\text{BICC}} \times N_y/M_{\text{BICC}} \) and \( N_x/M_{\text{BICC}} \times N_y/M_{\text{BICC}} \).

**Worker computation phase:** The \( n \)-th worker first computes \( C((n-1)P+1) = \hat{A}((n-1)P+1)B((n-1)P+1) \) and transmits the result \( C((n-1)P+1) \) back to the master. That same worker next computes \( C((n-1)P+2) = \hat{A}((n-1)P+2)B((n-1)P+2) \) and sends the result to the master. Likewise, it sequentially completes subtasks up to \( P \) subtasks, transmitting each result to the master. The transmission of partial (per-sub-task) results is a novel aspect of BICC and is an essential aspect required to exploit the work performed by all workers.

**Result aggregation phase:** The master receives \( N_x/M_{\text{BICC}} \times N_y/M_{\text{BICC}} \) matrices sequentially from all workers till it receives at least \( R_{\text{BICC}} \) distinct completed subtasks.

**Decoding phase:** Once the master collects \( R_{\text{BICC}} \) completed subtasks, it starts decoding. The decoding algorithm depends on the code. If the master uses polynomial or MatDot codes in the encoding phase, the master can use a polynomial interpolation algorithm or, equivalently, a Reed-Solomon decoder [27].

### Table II: Parameter decision in different methods of hierarchical coding and non-hierarchical schemes.

| Block length | Non-h | BICC | MLCC | HHCC |
|--------------|-------|------|------|------|
| Information dimension | \( K \) | \( K_{\text{BICC}} = KP \) | \( K_1, \sum_{i=1}^{P} K_i = K_{\text{sum}} = KP \) | \( K_{1,\text{HHCC}}, \sum_{i=1}^{L_{\text{HHCC}}} K_i = KP \) |
| Recovery threshold | \( R \) | \( R_{\text{BICC}} \) | \( R_1, \sum_{i=1}^{P} R_i = RP \) | \( R_{1,\text{HHCC}}, \sum_{i=1}^{L_{\text{HHCC}}} R_i = RP \) |
| # Levels | 1 | 1 | \( L = P \) | \( L_{\text{HHCC}} \in [1, P] \) |
| # Subtasks per level | \( P \) | \( P_{1,\text{MLCC}} = 1, \sum_{i=1}^{P} P_{i,\text{MLCC}} = P \) | \( P_{l} \in [1, P], \sum_{i=1}^{L_{\text{MLCC}}} P_{i} = P \) |

### Table III: Summary of parameters used in this paper.

![Toy examples of non-hierarchical and hierarchical coded computation](image)

**Fig. 4:** Toy examples of non-hierarchical and hierarchical coded computation, where we need 2 out of 3 computations to be completed.
task block \( l \in [L] \) can be visualized as a cuboid of dimensions \( N_{x_l} \times N_{y_l} \times N_{z_l} \), where

\[
\sum_{l \in [L]} N_{x_l} N_{y_l} N_{z_l} = N_x N_z N_y. \tag{5}
\]

The above equality holds by the assumption of disjoint task blocks. Partitioning the cuboid into task blocks is equivalent to dividing the matrix \( A \) into \( L \) matrices \( \{A_{S_{x_l} \times S_{z_l}} | l \in [L]\} \) and dividing the matrix \( B \) into \( L \) matrices \( \{B_{S_{z_l} \times S_{x_l}} | l \in [L]\} \), where \( N_{x_l} = |S_{x_l}| \), \( N_{z_l} = |S_{z_l}| \), and \( N_{y_l} = |S_{y_l}| \). Through this first step of partitioning the master decomposes the \( AB \) product into \( L \) computations \( A_{S_{x_l} \times S_{z_l}} B_{S_{z_l} \times S_{x_l}} \).

In the second partitioning step the master subvides the task block \( l \in [L] \) into \( K_l = M_{x_l} M_{z_l} M_{y_l} \) equal-sized information blocks each of dimensions \( N_{x_l}/M_{x_l} \times N_{z_l}/M_{z_l} \times N_{y_l}/M_{y_l} \). Through this partitioning phase, the master partitions \( A_{S_{x_l} \times S_{z_l}} \) into \( M_{x_l} \times M_{z_l} \) equal-sized matrices denoted by \( A_{m_{x_l}, m_{z_l}}^{(l)} \), where \( (m_{x_l}, m_{z_l}) \in [M_{x_l}] \times [M_{z_l}] \) and \( l \in [L] \). Likewise the matrices \( B_{S_{z_l} \times S_{x_l}} \) are partitioned into \( M_{z_l} \times M_{y_l} \) equal-sized matrices \( B_{m_{z_l}, m_{y_l}}^{(l)} \), where \( (m_{z_l}, m_{y_l}) \in [M_{z_l}] \times [M_{y_l}] \) and \( l \in [L] \). Note that the matrix product \( A_{S_{x_l} \times S_{z_l}} B_{S_{z_l} \times S_{x_l}} \) for the \( l \) task block is partitioned into \( K_l \) matrix multiplication \( A_{m_{x_l}, m_{z_l}}^{(l)} B_{m_{z_l}, m_{y_l}}^{(l)} \).

Two comments are in order. First, we assume that \( N_{x_l}, N_{z_l} \) and \( N_{y_l} \) are, respectively, much larger than \( M_{x_l}, \) \( M_{z_l}, \) and \( M_{y_l} \). This assumption allows us to ignore integer effects. Second, if \( v_l \) denotes the integer volume of the \( l \)th task block, i.e., \( v_l = N_{x_l} N_{y_l} N_{z_l} \), we choose \( M_{x_l}, M_{z_l}, \) and \( M_{y_l} \) so that \( v_l/(M_{x_l} M_{z_l} M_{y_l}) \) is (approximately) constant for all \( l \in [L] \). We need not make this choice, we make it to keep the quantity of computation constant across levels. The implication is that information blocks will be of (approximately) constant volume. In particular, we choose there to be \( K_{sum} = \sum_{l \in [L]} K_l \) information blocks each of (approximate) volume \( N_{x_l} N_{y_l} N_{z_l}/K_{sum} \). This assumption will prove useful when computing the response times of workers and when comparing to the results of previous papers. We note that the assumption that we keep \( v_l/(M_{x_l} M_{z_l} M_{y_l}) \approx N_{x_l} N_{z_l} N_{y_l}/K_{sum} \) constant does not mean that the height, width and depth of information blocks must be the same across different levels. Only the volume of information blocks is kept constant. The volume corresponds to the number of basic operations in each computation and thus each subtask is an equal amount of work.

**Encoding phase:** For each level \( l \in [L] \), the master generates \( N \) pairs of encoded matrices \( (A_i(n), B_i(n)) \), \( n \in [N] \), from the \( 2K_l \) matrices \( A_{m_{x_l}, m_{z_l}}^{(l)} \) and \( B_{m_{z_l}, m_{y_l}}^{(l)} \), where \((m_{x_l}, m_{z_l}, m_{y_l}) \in [M_{x_l}] \times [M_{z_l}] \times [M_{y_l}] \). To do this, the master applies coding across the matrices \( A_{m_{x_l}, m_{z_l}} \) and (separately) \( B_{m_{z_l}, m_{y_l}}^{(l)} \) to generate \( L \) encoded matrices for worker \( n \in [N] \). \( L \) distinct codes are used. For instance, by using polynomial codes [9], the master evaluates \( L \) pairs of polynomials \( (A_l(x), B_l(x)) \), \( l \in [L] \), at \( x = n \) to generates encoded matrices for worker \( n \in [N] \).

**Distribution phase:** The master sends \( P = L \) pairs of encoded matrices to each worker. For the \( n \)th worker this is the set \( \{(A_l(n), B_l(n)) | l \in [L]\} \). Note that, for \( l \in [L] \), the encoded matrices \( A_i(n) \) and \( B_i(n) \) are, respectively, of dimensions \( N_{x_l}/M_{x_l} \times N_{z_l}/M_{z_l} \) and \( N_{y_l}/M_{y_l} \times N_{z_l}/M_{z_l} \). The worker sequentially computes its \( P = L \) subtasks, \( A_1(n) B_1(n) \) through \( A_L(n) B_L(n) \). Each worker sends each completed subtask to the master as soon as it is finished.

**Result aggregation phase:** To recover all the information blocks that make up the \( l \)th level of computation (and thus to recover the \( l \)th task block), the master must receive at least \( R_l \) jobs from the \( N \) workers. Any subset of cardinality at least \( R_l \) of \( \{A_i(n), B_i(n) | n \in [N]\} \) will suffice. We term the choice of per-level recovery thresholds \( \{R_l\}_{l \in [L]} \) the recovery profile. The recovery profile is something that we optimize in Sec. VII based on the statistics of the distribution of processing times. Furthermore, in MLCC the same amount of computation compared to BICC is required to be completed, i.e., \( \sum_{l=1}^{L} R_l = LR \).

**Decoding phase:** Due to the independence encoding of each level, the decoding phase of MLCC can be carried out either in a serial, a parallel, or a streaming manner across levels. In serial and parallel decoding the master starts decoding when it receives enough completed jobs from all levels. In serial decoding the master pipelines the decoding. In parallel decoder, decoding of all levels is run in parallel after that the computation of all levels is finished. In streaming decoding the master starts decoding each level once it receives enough completed jobs of that specific level. The master doesn’t need to wait for all levels to be finished.

**Remark 1:** We note that uniform randomly shuffling the order of levels which workers work through can mitigate the need to design an optimal recovery profile in MLCC. This is indeed a special case of MLCC in which workers randomly pick an encoded subtask (without replacement) and sends its result to the master as soon as it is completed. We term this approach randomized MLCC (RMLCC). In RMLCC, we set the recovery profile to be \( R_l = R \) for all \( l \in [L] \), thus mitigating the need to profile design. The intuition behind this setting is because of the fact that for \( N \gg L \), on average an equal-sized group of workers would try different permutations of orders. Therefore, each level would get the same attention. This makes the expected finishing time of levels to be approximately the same without a need to design the recovery profile.

**C. Hybrid hierarchical coded matrix multiplication**

We now introduce HHCC as a general hierarchical coding approach, with BICC and MLCC as two special cases. Given \( (L_{HHCC}, R_{HHCC}, P, N, R_{HHCC}) \), we next detail HHCC.

**Data partitioning phase and cuboid visualization:** In HHCC the master partitions the cuboid in two steps. In the first step the master partitions the cuboid into \( L_{HHCC} \) heterogeneously sized task blocks, the \( l \)th of which is of dimensions \( N_{x_l}/L_{HHCC} \times N_{z_l}/L_{HHCC} \times N_{y_l}/L_{HHCC} \). The parameter \( L_{HHCC} \) is larger than the number of levels in BICC, which is \( L_{BICC} = 1 \), and smaller than the number of levels in MLCC, which is \( L_{MLCC} = P \). Therefore, the number of levels in
HHCC is somewhere between those of MLCC and BICC, i.e., \( L_{HHCC} \in [P] \). Assuming that task blocks are disjoint yields a similar constraint to (5), i.e.,

\[
\sum_{l \in [L_{HHCC}]} N_{zl,HHCC}N_{zl,HHCC}N_{yl,HHCC} = N_xN_yN_y,
\]

(7)

Through the first step of partitioning the matrix product \( AB \) is divided into \( L_{HHCC} \) computations \( A_{S_{l1} \times S_{l2}}B_{S_{l1} \times S_{l2}} \), where \( l \in [L_{HHCC}] \), \( |S_{z1}| = N_{zl,HHCC} \), \( |S_{z1}| = N_{zl,HHCC} \), and \( |S_{yl}| = N_{yl,HHCC} \).

In the second step the master subdivides the \( l \)th task block into \( K_{l,HHCC} = M_{zl,HHCC}M_{zl,HHCC}M_{yl,HHCC} \) equal-sized information blocks. To do this, the master cuts the cuboid with \( M_{zl,HHCC} - 1 \) slices along the \( x \)-axis, and \( M_{zl,HHCC} - 1 \) and \( M_{yl,HHCC} - 1 \) slices along \( z \)- and \( y \)-axes, respectively. Through the second step of partitioning the matrix \( A_{S_{l1} \times S_{l2}} \) is subdivided into \( M_{zl,HHCC}M_{zl,HHCC} \) equal-sized matrices \( A_{m_{zl},m_{zl},m_{zl},m_{zl}} \), where \( (m_{zl}, m_{zl}) \in [M_{zl,HHCC}] \times [M_{zl,HHCC}] \) and \( l \in [L_{HHCC}] \). Likewise, each of \( B_{S_{l1} \times S_{l2}} \) matrices, \( l \in [L_{HHCC}] \), is subdivided into \( M_{zl,HHCC}M_{zl,HHCC} \) equal-sized matrices \( B_{m_{zl},m_{zl},m_{zl},m_{zl}} \), where \( (m_{zl}, m_{zl}) \in [M_{zl,HHCC}] \times [M_{zl,HHCC}] \). At the end of the partitioning phase, the matrix product \( A_{S_{l1} \times S_{l2}}B_{S_{l1} \times S_{l2}} \) is partitioned into \( K_{l,HHCC} \) multiplications of \( A_{m_{zl},m_{zl},m_{zl},m_{zl}} \) and \( B_{m_{zl},m_{zl},m_{zl},m_{zl}} \) matrices, where \( (m_{zl}, m_{zl}) \in [M_{zl,HHCC}] \times [M_{zl,HHCC}] \times [M_{zl,HHCC}] \). Note that the \( A_{m_{zl},m_{zl},m_{zl},m_{zl}} \) and \( B_{m_{zl},m_{zl},m_{zl},m_{zl}} \) matrices are, respectively, of dimensions \( N_{zl,HHCC}/M_{zl,HHCC} \times N_{zl,HHCC}/M_{zl,HHCC}/M_{zl,HHCC} \) and \( N_{zl,HHCC}/M_{zl,HHCC} \times N_{zl,HHCC}/M_{zl,HHCC}/M_{zl,HHCC} \).

**Encoding phase:** For each \( l \in [L_{HHCC}] \), the master encodes \( 2K_{l} \) matrices \( A_{m_{zl},m_{zl},m_{zl},m_{zl}} \) and \( B_{m_{zl},m_{zl},m_{zl},m_{zl}} \) to generate \( P \) pairs of encoded matrices \( \{A_l(n), B_l(n)\} \), \( n \in [P] \). We use \( P \) to denote the number of subtasks each worker is assigned to contribute to the completion of the \( l \)th level of computation. In MLCC we have \( P_l = 1 \) for all \( l \in [L] \). On the other hand, in BICC we have only one level \((L_{BCICC} = 1)\), thus, \( P_l = P \). If we again follow [9], the master uses polynomial codes to generate encoded matrices. For level \( l \in [L_{HHCC}] \), the master evaluates polynomials \( \tilde{A}_l(x) \) and \( \tilde{B}_l(x) \) at \( P_l \) distinct points \( x \in \{(n - 1)P_l + 1, \ldots, (n - 1)P_l + P_l\} \) to generate encoded matrices for worker \( n \in [N] \).

**Distribution phase:** The master next sends \( P \) pairs of encoded submatrices to each worker, one pair per level, \( l \in [L_{HHCC}] \). We note that in total, the master sends \( P \) pairs of encoded submatrices to each worker, where

\[
P = \sum_{l \in [L_{HHCC}]} P_l.
\]

The master is required to send all encoded matrices \( \tilde{A}_l((n - 1)P_l + p) \) and \( \tilde{B}_l((n - 1)P_l + p) \) to worker \( n \in [N] \), for all \( l \in [L_{HHCC}] \) and \( p \in [P] \). We note that the \( \tilde{A}_l((n - 1)P_l + p) \) and \( \tilde{B}_l((n - 1)P_l + p) \) matrices are, respectively, of dimensions \( N_{zl,HHCC}/M_{zl,HHCC} \times N_{zl,HHCC}/M_{zl,HHCC}/M_{zl,HHCC} \) and \( N_{zl,HHCC}/M_{zl,HHCC} \times N_{yl,HHCC}/M_{yl,HHCC} \).

**Worker computation phase:** Similar to MLCC and BICC, in HHCC each worker sequentially completes its \( P \) subtasks and transmits each result to the master as soon as each is completed. This means that, worker \( n \in [N] \) first completes a sequence of \( P_l \) matrix multiplications \( \{\tilde{A}_l((n - 1)P_l + 1)\tilde{B}_l((n - 1)P_l + 1), \ldots, \tilde{A}_l((n - 1)P_l + 1)\tilde{B}_l((n - 1)P_l + P_l)\} \). It then performs \( P_2 \) matrix multiplications pertinent to second level, i.e., \( \tilde{A}_2((n - 1)P_2 + p)\tilde{B}_2((n - 1)P_2 + p) \), where \( p \in [P_2] \). It continues to sequentially multiply matrices up to completion of \( \tilde{A}_L(\tilde{B}_L(n - 1)P_{L_{HHCC}} + 1)\tilde{B}_L((n - 1)P_{L_{HHCC}} + 1) \) through \( \tilde{A}_L(\tilde{B}_L((n - 1)P_{L_{HHCC}} + P_{L_{HHCC}})\tilde{B}_L((n - 1)P_{L_{HHCC}} + P_{L_{HHCC}}) \).

**Result aggregation phase:** The master receives subtasks sequentially from all workers till it receives \( R_{L_{HHCC}} \) completed subtasks for the \( l \)th level. For each \( l \in [L_{HHCC}] \), any subset of cardinality at least \( R_{L_{HHCC}} \) of \( \{\tilde{A}_l(n)\tilde{B}_l(n)\}n \in [P]\) is required. In HHCC \( \sum_{l=1}^{L_{HHCC}} R_l \approx PR \).

**Decoding phase:** Once the master collects enough completed subtasks for level \( l \), it starts decoding all subtasks pertinent to that level. In MLCC decoding can be parallelized, while in BICC the subtasks of all workers are decoded jointly as part of a single code. The decoding algorithm the master uses in this phase depends on the coding approach the master used in the encoding phase. For instance, in the case of using polynomial or MatDot codes, the master uses a polynomial interpolation decoder, while in the case of product codes, the master uses a simple peeling decoder.

**D. Examples**

To illustrate our results we use polynomial codes [9] to present different hierarchical coded computing approaches. In the following four examples we first detail the usage of polynomial codes [9] in BICC. We then present in detail MLCC when polynomial coding is used for each level. For illustrative reasons, in both of these examples we assume that \( P = 4 \) subtasks are assigned to each worker. The computation load of \( n \)th worker is kept constant at \( C_{\text{comp}} = \frac{N_xN_yN_y}{PP} \) to enable a fair comparison. In the last two examples we use polynomial codes in HHCC. In Example 5 we use HHCC with \( L_{HHCC} = 2 \), while in Example 6 we use \( L_{HHCC} = 3 \).

**Example 3 (Bit-interleaved polynomial coding):** Similar to polynomial coding in Example 1, bit-interleaved polynomial coding has a one-step cuboid partitioning. This means that the master partitions the cuboid into \( K_{BCICC} = 16 \) equal-sized information blocks. The partitioning used in this example is depicted in Fig. [5]. Note that the information dimension \( K_{BCICC} \) of this code is \( P = 4 \) times larger than that of the polynomial codes (which had \( K_{\text{Poly}} = 4 \)). In other words, \( K_{BCICC} = PK_{\text{Poly}} \). To achieve the information dimension of \( K_{BCICC} = 16 \), the master divides each of the \( A \) and \( B \) matrices into four equal-sized submatrices. Respectively, these are

\[
A^T = [\tilde{A}_1^T, \tilde{A}_2^T, \tilde{A}_3^T, \tilde{A}_4^T],
B = [\tilde{B}_1, \tilde{B}_2, \tilde{B}_3, \tilde{B}_4],
\]

where \( \tilde{A}_i \in \mathbb{R}^{N_x \times N_y} \) and \( \tilde{B}_i \in \mathbb{R}^{N_y \times N_x} \) for \( i \in [4] \).

After partitioning, the master encodes the \( \tilde{A}_i \) and the \( \tilde{B}_i \) separately using polynomial codes, to generate encoded submatrices

\[
\tilde{A}(x) = \tilde{A}_1 + \tilde{A}_2x + \tilde{A}_3x^2 + \tilde{A}_4x^3,
\tilde{B}(x) = \tilde{B}_1 + \tilde{B}_2x^4 + \tilde{B}_3x^8 + \tilde{B}_4x^{12}.
\]
Worker $n$ is then tasked to compute $P = 4$ sequentially-ordered subtasks: \( \{ A(4n + i)B(4n + i) \mid i \in [4] \} \). Due to the use of polynomial codes, the completion of any $R_{\text{HHC}} = 16$ subtasks enables the recovery of the $AB$ product.

**Example 4 (Multilevel polynomial coding):** In this example each worker is provided $L_{\text{MLCC}} = 4$ sequentially-ordered subtasks. The master first divides the cuboid into $L_{\text{MLCC}} = 4$ task blocks of dimensions \( \left( N_{x_1}, N_{z_1}, N_{y_1} \right) \in \{(N_x, N_z, N_y), (N_x, \frac{N_y}{2}, N_y), \left( \frac{N_x}{2}, N_z, \frac{N_y}{2} \right), \left( \frac{N_x}{2}, \frac{N_z}{2}, \frac{N_y}{2} \right) \} \). It then subdivides each task block into $K_l \in \{4, 3, 1\}$ information blocks each of which contains $N_x N_z N_y / 16$ basic operations. In Fig. 5d the decomposition of the $AB$ product into levels of computation is depicted by the solid (blue) lines. The partitioning of task blocks into information blocks is indicated by the dashed lines. In Fig. 5d the projection of the information blocks and task blocks onto the $xy$ plane represents the decomposition of $AB$ into the computations shown in (8). For instance, the sub-computations of the first task block are the product of matrices $A^{(1)}_{m_x, m_y} \in \mathbb{R}^{N_x \times N_y}$ and $B^{(1)}_{m_x, m_y} \in \mathbb{R}^{N_x \times \frac{N_y}{2}}$, where $m_x \in [4], m_y \in [1]$ and $m_y \in [2]$.

\[
\begin{align*}
A &= \begin{bmatrix} A^{(1)}_{1,1} & A^{(1)}_{1,2} & A^{(2)}_{1,1} & A^{(2)}_{1,2} & A^{(3)}_{1,1} & A^{(3)}_{1,2} \\ A^{(1)}_{2,1} & A^{(1)}_{2,2} & A^{(2)}_{2,1} & A^{(2)}_{2,2} & A^{(3)}_{2,1} & A^{(3)}_{2,2} \\ A^{(1)}_{3,1} & A^{(1)}_{3,2} & A^{(2)}_{3,1} & A^{(2)}_{3,2} & A^{(3)}_{3,1} & A^{(3)}_{3,2} \\ A^{(1)}_{4,1} & A^{(1)}_{4,2} & A^{(2)}_{4,1} & A^{(2)}_{4,2} & A^{(3)}_{4,1} & A^{(3)}_{4,2} \end{bmatrix} \\
B &= \begin{bmatrix} A^{(1)}_{1,1} & A^{(1)}_{1,2} & A^{(2)}_{1,1} & A^{(2)}_{1,2} & A^{(3)}_{1,1} & A^{(3)}_{1,2} \\ A^{(1)}_{1,1} & A^{(1)}_{1,2} & A^{(2)}_{1,1} & A^{(2)}_{1,2} & A^{(3)}_{1,1} & A^{(3)}_{1,2} \\ A^{(1)}_{2,1} & A^{(1)}_{2,2} & A^{(2)}_{2,1} & A^{(2)}_{2,2} & A^{(3)}_{2,1} & A^{(3)}_{2,2} \\ A^{(1)}_{2,1} & A^{(1)}_{2,2} & A^{(2)}_{2,1} & A^{(2)}_{2,2} & A^{(3)}_{2,1} & A^{(3)}_{2,2} \\ A^{(1)}_{3,1} & A^{(1)}_{3,2} & A^{(2)}_{3,1} & A^{(2)}_{3,2} & A^{(3)}_{3,1} & A^{(3)}_{3,2} \\ A^{(1)}_{3,1} & A^{(1)}_{3,2} & A^{(2)}_{3,1} & A^{(2)}_{3,2} & A^{(3)}_{3,1} & A^{(3)}_{3,2} \\ A^{(1)}_{4,1} & A^{(1)}_{4,2} & A^{(2)}_{4,1} & A^{(2)}_{4,2} & A^{(3)}_{4,1} & A^{(3)}_{4,2} \end{bmatrix}
\end{align*}
\]

The master encodes the data relevant to the $l$th task block by applying a pair of polynomial codes (separately) to the matrices involved in that level. In the above example, the polynomials used to encode are

\[
\begin{align*}
\hat{A}_1(x) &= A^{(1)}_{1,1} + A^{(1)}_{2,1} x + A^{(1)}_{3,1} x^2 + A^{(1)}_{4,1} x^3, \\
\hat{B}_1(x) &= B^{(1)}_{1,1} + B^{(1)}_{1,2} x^4, \\
\hat{A}_2(x) &= A^{(2)}_{1,1} + A^{(2)}_{2,1} x + A^{(2)}_{3,1} x^2 + A^{(2)}_{4,1} x^3, \\
\hat{B}_2(x) &= B^{(2)}_{1,1}, \\
\hat{A}_3(x) &= A^{(3)}_{1,1} + A^{(3)}_{2,1} x + A^{(3)}_{3,1} x^2, \\
\hat{B}_3(x) &= B^{(3)}_{1,1}, \\
\hat{A}_4(x) &= A^{(4)}_{1,1}, \\
\hat{B}_4(x) &= B^{(4)}_{1,1}.
\end{align*}
\]
The master next encodes the computation of two task blocks using two independent bit-interleaved polynomial codes. For instance, the master can use two bit-interleaved polynomial codes, one with parameter set \((R_{BICC,1} = 12, P_1 = 3)\) for the first level and a bit-interleaved polynomial code with \((R_{BICC,2} = 4, P_2 = 1)\) for the second level. Therefore, the polynomials used to encode are
\[
\hat{A}_1(x) = A_{1,1}^{(1)} + A_{2,1}^{(1)}x + A_{3,1}^{(1)}x^2 + A_{4,1}^{(1)}x^3, \\
\hat{B}_1(x) = B_{1,1}^{(1)} + B_{1,2}^{(1)}x + B_{1,3}^{(1)}x^2, \\
\hat{A}_2(x) = A_{1,2}^{(2)} + A_{2,1}^{(2)}x + A_{3,1}^{(2)}x^2, \\
\hat{B}_2(x) = B_{1,2}^{(2)} + B_{1,3}^{(2)}x. 
\]

Worker \(n \in [N]\) receives \(P = 4\) pairs of encoded matrices \(\{A_1(3n + 1), B_1(3n + 1)\}, \{A_1(3n + 2), B_1(3n + 2)\}, \{A_2(3n + 3), B_2(3n + 3)\}, \{A_2(3n + 1), B_2(3n + 1)\}\) and sequentially multiplies the matrices in each pair. The master is able to recover the AB product when it receives 12 completed subtasks from the set \(\{A_1(n), B_1(n)\} n \in [3N]\) and 4 from the set \(\{A_2(n), B_2(n)\} n \in [N]\).

Example 6 (HHCC \((L_{HHCC} = 3, R_{HHCC} \in \{8, 6, 2\})\): In this example the master first uses a three-level multilevel polynomial code with profile \((8, 6, 2)\). It divides the cuboid into \(L_{HHCC} = 3\) task blocks and then subdivides the \(l\)th task block into \(K_{HHHC} \in \{8, 6, 2\}\) information blocks. The partitioning used in this example is depicted in Fig. 5C. Through this partitioning the matrix product AB is partitioned into 16 equal-sized subcomputations
\[
\begin{bmatrix}
A_{1,1}^{(1)} B_{1,1}^{(1)} & A_{1,1}^{(1)} B_{1,2}^{(1)} & A_{1,2}^{(1)} B_{1,1}^{(1)} & A_{1,2}^{(1)} B_{1,2}^{(1)} \\
A_{2,1}^{(1)} B_{1,1}^{(1)} & A_{2,1}^{(1)} B_{1,2}^{(1)} & A_{2,2}^{(1)} B_{1,1}^{(1)} & A_{2,2}^{(1)} B_{1,2}^{(1)} \\
A_{3,1}^{(1)} B_{1,1}^{(1)} & A_{3,1}^{(1)} B_{1,2}^{(1)} & A_{3,2}^{(1)} B_{1,1}^{(1)} & A_{3,2}^{(1)} B_{1,2}^{(1)} \\
A_{4,1}^{(1)} B_{1,1}^{(1)} & A_{4,1}^{(1)} B_{1,2}^{(1)} & A_{4,2}^{(1)} B_{1,1}^{(1)} & A_{4,2}^{(1)} B_{1,2}^{(1)}
\end{bmatrix}
\]
The master next encodes the computation of the three task blocks using three independent bit-interleaved polynomial codes using following parameters: for the first, \((R_{BICC,1} = 8, P_1 = 2)\), for the second \((R_{BICC,2} = 6, P_2 = 1)\), and for the third, \((R_{BICC,3} = 2, P_2 = 1)\). To generate encoded matrices, the master uses the polynomials
\[
\hat{A}_1(x) = A_{1,1}^{(1)} + A_{2,1}^{(1)}x + A_{3,1}^{(1)}x^2 + A_{4,1}^{(1)}x^3, \\
\hat{B}_1(x) = B_{1,1}^{(1)} + B_{1,2}^{(1)}x, \\
\hat{A}_2(x) = A_{1,2}^{(2)} + A_{2,1}^{(2)}x + A_{3,1}^{(2)}x^2, \\
\hat{B}_2(x) = B_{1,2}^{(2)} + B_{1,3}^{(2)}x, \\
\hat{A}_3(x) = A_{3,1}^{(3)} + A_{3,2}^{(3)}x, \\
\hat{B}_3(x) = B_{1,2}^{(3)} + B_{1,3}^{(3)}x. 
\]

Worker \(n \in [N]\) is tasked with a sequence of \(P = 4\) matrix products \(\hat{A}_1(2n + 1)\hat{B}_1(2n + 1), \hat{A}_1(2n + 2)\hat{B}_1(2n + 2), \hat{A}_2(n)\hat{B}_2(n), \) and \(\hat{A}_3(n)\hat{B}_3(n)\). The master is able to recover the AB product if it receives 12 completed computations from the set \(\{A_1(n)B_1(n)|n \in [2N]\}\), 6 completed computations from the set \(\{A_2(n)B_2(n)|n \in [N]\}\), and 2 completed computations from the set \(\{A_3(n)B_3(n)|n \in [N]\}\).

E. Discussion

To compare MLCC and BICC in Examples 3 and 4, we recall that each worker is tasked with the same number and size of subtasks. In those examples one can observe that BICC has a more flexible recovery rule than MLCC. While for BICC, the AB product can be recovered from any \(R_{BICC}\) completed subtasks, in MLCC the completed subtasks must follow a specific profile \(\{R_l\}\). On the other hand, from a decoding perspective, MLCC is much more complex than BICC. In BICC the master needs to deal with decoding \(N_2N_{3}/16\) polynomials of degree 15 when using polynomial codes. On the other hand, in MLCC the master is required to decode \(L_{MLCC} = 4\) sets of polynomials of (in the example) degrees 7, 3, 2, and 0, each set consisting of \(N_2N_{3}/16\) polynomials. As was discussed in Sec. IV-B in MLCC the master can perform such decoding either in a serial, a parallel, or a streaming manner across levels. Parallel and streaming decoding are not possible for BICC. In the numerical results of Sec. VII we observe that even serial decoding of MLCC takes less time than decoding BICC.

As would be expected, the parallel decoding time of MLCC is much less than the decoding time of BICC. This is due to the fact that in the decoding phase of multilevel polynomial codes, in a worst-case scenario, the master needs to deal with decoding a polynomial code of rate \(8/N\). This is much less computationally intensive than the decoding of the rate \(16/N\) polynomial code used in bit-interleaved polynomial codes. Streaming decoding takes the least time of all. Furthermore, in next section we design MLCC in such a way that they also outperforms BICC in terms of communication time. Compared to BICC, the difference of MLCC follows from the distinct rates applied across the levels, \(8/N, 4/N, 2/N, \) and \(1/N\) in the multilevel polynomial coding example and \(R_l/N\) in general. In BICC the sub-computations are encoded jointly as a part of a single code with the code rate \(R_{BICC}/(NP)\).

One way of conceiving of the difference between BICC and MLCC is by analogy with the two coded modulation techniques: BICM (bit-interleaved coded modulation) and MLCC (multilevel coding) [25]. In BICM a vector of encoded bits is first generated from information bits using a binary encoder and then is permuted using a bit-level interleaver. This single interleaved vector is then passed to the mapper. In contrast, an alternative technique is MLCC in which the information bits are first separated into multiple smaller message vectors. Next, each message vector is independently encoded and is passed to the mapper. The mapper receives multiple encoded bits in MLCC, while in BICM a single vector of interleaved bits is passed to the mapper. This distinction is analogous to that of BICC and MLCC. We use the term bit-interleaved and multilevel for our proposed coded computation schemes to highlight such parallels.

Now consider HHCC, as generalization and unification of BICC and MLCC. HHCC provides a tradeoff between the time to compute (recovery flexibility rule) and the times to communicate and to decode. It falls between two extreme cases of hierarchical coding: BICC and MLCC. This means...
that it has a lower communication and decoding times when compared to BICC, and larger communication and decoding times when compared to MLCC. On the other hand, the recovery conditions of HHCC are much more flexible than MLCC, while the recovery conditions of BICC are more flexible still.

**Remark 2:** In general we can use different coding schemes across different levels of computations. For instance we can use polynomial codes [9] to encode data in the first level, and use MatDot codes [10] for the second level. The flexibility to use different types of code at different levels of hierarchical coding can be considered for future work. In this paper we focus on single-type hierarchical coding. We also note that the idea of hierarchical coding is not limited to polynomial codes. For instance, one can apply hierarchical coding to product codes and repetition codes by leveraging their corresponding cuboid partitioning structures as is discussed in Sec. [III-B] In Sec. [VI] we apply hierarchical coding for other types of codes.

**V. Theoretical analysis**

In this section we use the performance measures introduced in Sec. [II] to determine the finishing times of hierarchical coding, and compare it to non-hierarchical coding. To accomplish this we assert probabilistic models on the worker computation and communication times. We use shifted exponential models for analytic tractability. We compute the expected finishing times and optimize the parameters of the schemes to minimize that time.

**A. Finishing time formulation**

We term the time it takes the system to compute the AB product the **finishing time**. A number of cumulative effects contribute to the finishing time: (i) the time to encode the data, (ii) the time to distribute the encoded data, (iii) the computation time required by the workers, (iv) the time to aggregate completed tasks, and (v) the time to decode. As encoding time is generally negligible when compared to computation time, we ignore (i) in the following. We denote the finishing time of the non-hierarchical, MLCC, and BICC, as $\tau_{Non-h}$, $\tau_{MLCC}$, and $\tau_{BICC}$, respectively.

**Non-hierarchical scheme:** Recall from Sec. [III] that the time to distribute to the nth worker the encoded data is $C_{out}^n T_{comp}^n$ sec. The time for that worker to complete its task is $C_{comp}^n T_{comp}^n$ sec. And the time for that worker to transmit its results to the master is $C_{out}^n T_{comp}^n$ sec. Therefore, the finishing time of worker $n$ is the random variable

$$T_n = C_{out}^n T_{comp}^n + C_{comp}^n T_{comp}^n + C_{out}^n T_{comp}^n.$$  \[12\]

In non-hierarchical schemes, the master is required to collect (at least) $R$ completed tasks. The time required is $T_{(R,N)}$, where $(\cdot)_{(R,N)}$ is an order-statistic operation that selects the $R$th element of the (sorted) $N$-element sequence $(T_n)_{n\in[N]}$. Let $\{i_1, \ldots, i_R\} \subseteq \{1, \ldots, N\}$ denote the indices of the sorted sequence $T_{i_1} \leq T_{i_2} \leq \ldots \leq T_{i_R}$. The outcome of $T_{(R,N)}$ is derived at the index $R^* = i_R$, i.e.,

$$T_{(R,N)} = T_{R^*}. \text{ The master then spends } T_{dec,non-h} \text{ sec to decode yielding a total finishing time of }$$

$$\tau_{non-h} = T_{R^*} + T_{dec,non-h}. \quad \text{(13)}$$

We further expand [12] by substituting in expressions for $C_{in}^n$, $C_{comp}^n$, and $C_{out}^n$. The encoded matrices are of dimensions $\hat{A}(n) \in \mathbb{R}^{M_x \times M_y}$ and $\hat{B}(n) \in \mathbb{R}^{M_x \times M_y}$ where $K = M_x M_y M_y$ so

$$C_{in}^n = N_x N_z + \frac{N_y N_z}{M_x M_y}.$$  \[14\]

The computation of $\hat{A}(n)\hat{B}(n)$ requires

$$C_{comp}^n = \frac{N_x N_z N_y}{M_x M_y M_z}.$$  \[15\]

The (encoded) completed result is of size $\frac{N_x}{M_x} \times \frac{N_y}{M_y}$, so

$$C_{out}^n = \frac{N_x N_y}{M_x M_y}.$$  \[16\]

Using [14]–[16] and [12] in [13], an expression for the finishing time of non-hierarchical schemes is

$$\tau_{non-h} = T_{R^*} \left( C_{in}^n + \frac{N_y N_z}{M_x M_y M_z} \right) + T_{dec,non-h}.$$  \[17\]

This is an order statistic amenable to analysis. Note that in [17] the $(\cdot)_{(R,N)}$ operation is applied into a sequence of weighted sum of $T_{comp}^n$ and $T_{comp}^n$ $n \in [N]$. This is not necessarily equal to weighted sum of $T_{comp}^{(R,N)}$ and $T_{comp}^{(R,N)}$ where $T_{comp}^{(R,N)}$ and $T_{comp}^{(R,N)}$ denote, respectively, the $R$th element of the sorted $N$-element sequences $(T_{comp}^n)_{n\in[N]}$ and $(T_{comp}^n)_{n\in[N]}$.

**Bit-interleaved coding scheme:** Recall that in BICC each worker is assigned $P$ subtasks. Further, these subtasks are tackled in order. We use $T_{(n-1)} P + P$ to denote the time worker $n$ takes to finish subtask $p \in [P]$. As different workers will complete different numbers of jobs we need a per-worker count of the number of subtasks provided to the master. The variable $p$ will play that role in the ensuing discussion. The master first spends $C_{n,i}^n T_{comp}^n$ sec to distribute the data pertinent to all $P$ (encoded) subtasks to worker $n$. That worker then spends $\sum_{j \in [i]} C_{n,j}^n T_{comp}^n$ sec to finish its first $i \in [p]$ subtasks, where $C_{n,j}^n$ denotes the computation load of the $j$th subtask of worker $n$. Each subtask is transmitted to the master upon completion. The $i$th such transmission takes $C_{out}^n T_{comp}^n$ seconds where $C_{out}^n$ is the output communication load of the $i$th subtask of worker $n$. In aggregate, to complete its first $p$ subtasks, worker $n$ requires

$$\max_{i \in [p]} \left[ C_{out}^n T_{comp}^n + \sum_{j \in [i]} C_{comp}^n T_{comp}^n \right] \text{ sec},$$

where the maximum, in general, may not be achieved by $i = p$.
due to the possibly lower $C_{n,i}^{\text{out}}$ of later layers. Thus,

$$\tilde{T}_{(n-1)P+p} = C_{n}^{-T} \tau_{\text{comm}} \max_{i\in[p]} \left[ C_{n,i}^{-T} \tau_{\text{comm}} + \sum_{j \in [i]} C_{n,j}^{-T} \tau_{\text{comm}} \right].$$

In BICC the master is able to recover $AB$ when it has received a total of at least $R_{\text{BICC}}$ subtasks. The finishing time of BICC can therefore be written as

$$\tau_{\text{BICC}} = \tilde{T}_{R_{\text{BICC}}} + T_{\text{dec,BICC}}$$

where $\tilde{T}_{R_{\text{BICC}}}$ is equal to $\tilde{T}_{(R_{\text{BICC}}-PN)}$ and denotes the $R_{\text{BICC}}$th element of the sorted $PN$-element sequence $\{(\tilde{T}_{n})_{|_{n\in[P,N]}}\}$. As before, we now make the expressions for $C_{n,i}^{\text{in}}$, $C_{n,i}^{\text{out}}$, and $C_{n,j}^{\text{comp}}$ explicit. In BICC the master conveys $2P$ distinct encoded submatrices $\hat{A}_{i}((n-1)P+p) \in \mathbb{R}^{N \times N_{x} \times \hat{M}_{x,\text{BICC}}}$ and $\hat{B}_{i}((n-1)P+p) \in \mathbb{R}^{N \times N_{y} \times \hat{M}_{y,\text{BICC}}}$ where $p \in [P]$ to worker $n$. Thus,

$$C_{n,i}^{\text{in}} = \frac{PN_{x} N_{y}}{\hat{M}_{x,\text{BICC}} \hat{M}_{y,\text{BICC}}} + \frac{PN_{y} N_{y}}{\hat{M}_{y,\text{BICC}} \hat{M}_{y,\text{BICC}}}$$

The worker is tasked with $P$ subtasks, each consisting of

$$C_{n,i}^{\text{comp}} = \frac{N_{x} N_{y}}{\hat{M}_{x,\text{BICC}} \hat{M}_{y,\text{BICC}}} = \frac{N_{x} N_{y}}{\hat{P} M_{x} M_{y}}$$

basic multiply-and-accumulate operations. Note the last step follows because $K_{\text{BICC}} = PK$. Finally, the $i$th worker-to-master transmission contains

$$C_{n,i}^{\text{out}} = \frac{N_{y}}{\hat{M}_{x,\text{BICC}} \hat{M}_{y,\text{BICC}}}$$

entries. Note that the right-hand-sides of both (21) and (22) are independent of $n$ and $i$, thus the maximum term in (18) is achieved by $i = p$. From (21), we can also conclude that per-worker computation time is linear in the number of subtasks the worker completes. In other words, worker $n$ spends $\sum_{j \in [i]} C_{n,j}^{\text{comp}} \tau_{\text{comp}} = \tau_{\text{comp}} \frac{N_{x} N_{y}}{\hat{P} M_{x} M_{y}}$ sec to complete its first $i$ subtasks. Thus, we can simplify (18) to

$$\tilde{T}_{(n-1)P+p} = C_{n}^{-T} \tau_{\text{comm}} + C_{n,i}^{\text{out}} \tau_{\text{comm}} + p C_{n,p}^{\text{comp}} \tau_{\text{comp}}.$$  

Merging $C_{n,i}^{\text{in}}$, $C_{n,i}^{\text{comp}}$, and $C_{n,i}^{\text{out}}$ into (23) and (19) yields

$$\tau_{\text{BICC}} = \tau_{\text{comp}}^{-1} \left( \frac{PN_{x} N_{y}}{\hat{M}_{x,\text{BICC}} \hat{M}_{z,\text{BICC}}} + \frac{PN_{y} N_{y}}{\hat{M}_{y,\text{BICC}} \hat{M}_{y,\text{BICC}}} \right)$$

$$+ \frac{N_{x} N_{y}}{\hat{M}_{x,\text{BICC}} \hat{M}_{y,\text{BICC}}} + \tau_{\text{comp}}^{-1} \frac{p^{*} N_{x} N_{y}}{\hat{P} M_{x} M_{y}} + T_{\text{dec,BICC}},$$

where $n^{*}$ and $p^{*}$ correspond to the $R_{\text{BICC}}$th order statistics of $\{T_{n}\}_{|_{n\in[0,P,N]}}$ such that $(n^{*}-1)P+p^{*} = R_{\text{BICC}}$, $1 \leq n^{*} \leq N$, and $0 \leq p^{*} \leq P$.

**Multilevel coding scheme:** In MLCC the overall job of computing $AB$ completes when each of the $L$ layers completes. For layer $l$ to complete, at least $R_{l}$ workers must finish their $l$th subtask. We denote the finishing time of layer $l$ by worker $n$ as $\tilde{T}_{n,l}$, which can be written as

$$\tilde{T}_{n,l} = C_{n}^{-T} \tau_{\text{comm}} + \max_{i\in[l]} \left[ C_{n,i}^{-T} \tau_{\text{comm}} + \sum_{j \in [i]} C_{n,j}^{-T} \tau_{\text{comp}} \right].$$

In above expression $C_{n,i}^{\text{comp}}$ and $C_{n,j}^{\text{out}}$ are the layer-$l$ computation load and output communication load of worker $n$, similar to BICC. Note that (25) is similar to (18), except that in BICC when we define the per-subtask finishing time of all workers we use a single sequence, $\{(\tilde{T}_{n})_{|_{n\in[N]}}, p \in [P]\}$. In contrast, in MLCC we use $L$ distinct sequences, $\{(\tilde{T}_{n})_{|_{n\in[N]}}, \ldots, (\tilde{T}_{n})_{|_{n\in[N]}}\}$. We compute the finishing time of MLCC as

$$\tau_{\text{MLCC}} = \max_{i\in[L]} \left[ T_{l,1}^{i} \right] + T_{\text{dec,MLCC}},$$

where $T_{l,1}^{i}$ is equal to $\tilde{T}_{l,1}^{i}$ and denotes the $R_{l}$th order static of the sequence $\{(\tilde{T}_{n})_{|_{n\in[N]}}\}$.

As before, we now make the expressions for $C_{n,i}^{\text{in}}$, $C_{n,i}^{\text{out}}$, and $C_{n,j}^{\text{comp}}$ explicit. Worker $n$ requires all encoded matrices $\hat{A}_{i}(n) \in \mathbb{R}^{N_{x} \times N_{z} \times \hat{M}_{x,\text{BICC}}}$ and $\hat{B}_{i}(n) \in \mathbb{R}^{N_{y} \times N_{z} \times \hat{M}_{y,\text{BICC}}}$, $l \in [L]$. The worst case is when all $\{\hat{A}_{i}(n)\} l \in [L]$ and $\{\hat{B}_{i}(n)\} l \in [L]$ are distinct. In this case, the master conveys all $\{(\hat{A}_{i}(n), \hat{B}_{i}(n))\} l \in [L]$ to worker $n$. However, in Sec. V.C we make a particular choice of parameters $N_{x,l}, N_{y,l}, M_{x,l}, M_{y,l}, M_{gl}$ which yields a set of encoded matrices in which many elements are equal. In that situation the master is able to convey encoded matrices to workers by sending only a subset of matrices as representative of all elements. Through such a design, the total input communication load of worker $n$ can be reduced. In general, an upper bound is

$$C_{n}^{\text{in}} \leq \sum_{l=1}^{L} \left( \frac{N_{x,l} N_{z}}{M_{x,l} M_{y,l}} + \frac{N_{y,l} N_{z}}{M_{y,l} M_{y,l}} \right).$$

Each worker multiplies the encoded matrices $\hat{A}_{i}(n) \in \mathbb{R}^{N_{x} \times N_{z} \times \hat{M}_{x,\text{BICC}}}$ and $\hat{B}_{i}(n) \in \mathbb{R}^{N_{y} \times N_{z} \times \hat{M}_{y,\text{BICC}}}$ as its $l$th layer subtask. This requires $N_{x,l} N_{z} N_{gl}/(M_{x,l} M_{y,l})$ basic operations. Therefore,

$$C_{n,l}^{\text{comp}} = \frac{N_{x,l} N_{y,l}}{M_{x,l} M_{y,l}} = \frac{N_{x,l} N_{y,l}}{K_{\text{sum}}}.$$
the master; hence,
\[ C_{n,l}^\text{out} = \frac{N_x N_y}{M_x M_y} \text{k.} \tag{30} \]

Incorporating \( C_{n,l}^\text{comp} \) and \( C_{n,l}^\text{out} \) into (29) and (26) yields a bound on the finishing time of MLCC:
\[
\tau_{\text{MLCC}}^{\text{MLCC}} \leq \max_{l \in [L]} \left[ T_{\text{comm}}^{\text{MLCC}} \sum_{i=1}^{L} \left( \frac{N_x N_z + N_y N_z}{M_x M_y} \right) + \max_{i=1}^{L} \left( T_{\text{comp}}^{\text{MLCC}} \frac{N_x N_y}{M_x M_y} + T_{\text{dec,MLCC}} \right) \right].
\tag{31}
\]

In Sec. V-C we design a MLCC scheme so that the computation load of each worker is at most equal to that of the non-hierarchical scheme, i.e., \( N_x N_z N_y / K_{\text{sum}} = N_x N_z N_y / M_x M_y M_z \). Solving for \( K_{\text{sum}} \) we get
\[ K_{\text{sum}} = L M_x M_y M_z. \tag{32} \]

At a fixed per-worker computation load, the MLCC scheme has \( C_{n,l}^\text{comp} \), and \( C_{n,l}^\text{out} \) no larger than those of the non-hierarchical schemes (14)-(16); i.e.,
\[
C_{n,l}^\text{comp} \leq \frac{N_x N_z}{M_x M_z} + \frac{N_y N_z}{M_y M_z},
\]
\[
C_{n,l}^\text{comp} \leq \frac{N_x N_y}{LM_x M_y M_z},
\]
\[
C_{n,l}^\text{out} \leq \frac{N_x N_y}{M_x M_y}.
\tag{33}
\]

Note that in all the above expressions, the right-hand-sides are independent of \( n \) and \( l \). In Sec. V-C we design a MLCC scheme such that the left-hand-sides of above expressions are also independent of \( n \) and \( l \). Thus, the maximum term in (29) is achieved by \( i = l \). This yields
\[ \tilde{T}_{n}^{\text{MLCC}} = C_{n,l}^\text{comp} + C_{n,l}^\text{out} + I_{C_{n,l}^\text{comp},T_{n}^{\text{MLCC}}} \tag{34} \]

Using (33) in (34) and (26) yields the following upper bound for finishing time of MLCC.
\[
\tau_{\text{MLCC}}^{\text{MLCC}} \leq \max_{l \in [L]} \left[ T_{\text{comp}}^{\text{MLCC}} \frac{N_x N_z}{M_x M_z} + T_{\text{dec,MLCC}} \right].
\tag{35}
\]

B. Probabilistic model

We now assign a random distribution model to the computation time \( T_{n}^{\text{comp}} \) and the communication time \( T_{n}^{\text{comm}} \). We denote \( F_{s}^{\text{comp}}(t) \) as the probability that a worker is able to finish \( s \) basic operations by time \( t \), i.e., \( F_{s}^{\text{comp}}(t) = \mathbb{P}(s T_{n}^{\text{comp}} \leq t) \). Similarly, \( T_{n}^{\text{comm}} \) is a random variable with distribution \( F_{\text{comm}}(t) = \mathbb{P}(s T_{n}^{\text{comm}} \leq t) \). \( F_{s}^{\text{comp}}(t) \) and \( F_{\text{comm}}(t) \) are assumed not to be a function of \( n, i.e., all workers are identical and therefore share independent yet identical distributed computation and communication times.

Computational model: In our analysis, we assume that workers complete tasks according to a shifted exponential distribution. The shifted exponential model is a widely used (e.g., in [6], [7]) model of computation time. Importantly, it provides design guidance in the choice of per-layer parameters such as the recovery profile. The shifted exponential distribution is parameterized by a scale parameter \( \mu \) and a shift parameter \( \alpha \).

We use a shifted exponential distribution with the parameter set \((\mu_{\text{comp}}, \alpha_{\text{comp}})\) to define the computation time \( T_{n}^{\text{comp}} \). \( T_{n}^{\text{comp}} \) is assumed to be conditionally deterministic. This means that the probability of a worker is able to finish \( s \) basic operations by time \( t \) satisfies
\[
F_{s}^{\text{comp}}(t) = 1 - e^{-\frac{t}{\mu_{\text{comp}}}} \left( \frac{\alpha_{\text{comp}}}{\mu_{\text{comp}}} \right), \quad \text{for } t \geq s \alpha_{\text{comp}}, \tag{36}
\]

which is a shifted exponential model. In other words, we assumed that workers make linear progress conditioned on the time it takes to compute one basic operation.

Communication model: Similar to the computation model, we use a shifted exponential distribution with parameter set \((\mu_{\text{comm}}, \alpha_{\text{comm}})\) to describe \( T_{n}^{\text{comm}} \), the time worker \( n \) takes to communicate an element of a matrix with the master node. The shifted exponential distribution we assert on communication time is motivated by the experiments we conducted on Amazon EC2. As demonstrated in App. [A] a shifted exponential distribution provides a good fit to the distribution of communication time. As noted before, we make a conditionally deterministic assumption: the probability that the \( n \)th worker is able to communicate a matrix of \( s \) entries to the master by time \( t \) is
\[
F_{s}^{\text{comm}}(t) = 1 - e^{-\frac{t}{\mu_{\text{comm}}}} \left( \frac{\alpha_{\text{comm}}}{\mu_{\text{comm}}} \right), \quad \text{for } t \geq s \alpha_{\text{comm}}. \tag{37}
\]

Regarding the above two probabilistic models, we make two further assumptions. First, we assume there are \( N \) distinct and independent routes between the master and the workers. This means that either the master or any worker is able to send its resulting matrices as soon as they are completed. Second, all workers are assumed to be statistically identical, having independent yet identical distributed computation and communication times. This models a homogeneous computation fabric. Such homogeneous assumption is easily relaxed to a heterogeneous one by assigning each worker specific \( \mu_{\text{comp}}, \alpha_{\text{comp}}, \mu_{\text{comm}} \) and \( \alpha_{\text{comm}} \) parameters.

C. Expected finishing time

In this section we ignore decoding time, assuming it is negligible when compared to computation and communication times. We note that we do consider decoding time in our results conducted on Amazon EC2 in Sec. VI to calculate expected finishing time, we consider two regimes: the fast-network and the fast-worker regimes. Each regime is determined by how the computation and communication times influence the total finishing time. The fast-network regime corresponds to a master-worker model where the network is fast, but workers are slow. This means that computation time plays a much more substantial role than does communication time. In the fast-worker regime the network is slow and the workers are fast. We next calculate the expected finishing time of non-hierarchical, BICC, and MLCC schemes.
Non-hierarchical coding scheme: To compute the expected finishing time of the non-hierarchical scheme, we take the expectation of (17). The expected finishing time is

\[ E[\tau_{\text{Non-h}}] = E[T_{\text{comp}}^{\text{comm}}] \left( \frac{N_x N_z}{M_{x}M_{z}} + \frac{N_x N_y}{M_{x}M_{y}} + \frac{N_y N_z}{M_{y}M_{z}} \right) + E[T_{\text{comp}}^{\text{rows}}] \left( \frac{N_x N_z N_y}{M_{x}M_{y}M_{z}} \right). \]

(38)

In the fast-network regime we ignore the communication term when we compute \( E[\tau_{\text{Non-h}}] \). Therefore, the \( R \)th order statistics of the sequence \( \{ T_n^{\text{comp}} \}_{n \in [N]} \) is obtained at index \( R^* \). From App. [C] we approximate \( E[\tau_{\text{Non-h}}] \) as

\[ E[\tau_{\text{Non-h}}] \approx \left( \sigma_{\text{comp}} + \mu_{\text{comp}} \log \left( \frac{N}{N - R} \right) \right) \left( \frac{N_x N_z N_y}{M_{x}M_{y}M_{z}} \right). \]

(39)

In the fast-worker regime the computation time is negligible when compared to communication time. Therefore, the \( R \)th order statistics of the sequence \( \{ T_n^{\text{comp}} \}_{n \in [N]} \) is obtained for index \( R^* \) in this regime. Using App. [C], we have

\[ E[\tau_{\text{Non-h}}] \approx \left( \sigma_{\text{comp}} + \mu_{\text{comp}} \log \left( \frac{N}{N - R} \right) \right) \left( \frac{N_x N_z N_y}{M_{x}M_{y}M_{z}} \right). \]

(40)

Bit-interleaved coding scheme: To calculate the expected finishing time of BICC, we take the expectation of (24).

\[ E[\tau_{\text{BICC}}] = E[T_{n^*}^{\text{comm}}] \left( \frac{PN_x N_z}{M_{x,BICC}M_{z,BICC}} + \frac{PN_x N_y}{M_{x,BICC}M_{y,BICC}} \right) + \frac{N_z N_y}{M_{x,BICC}M_{y,BICC}} + E\left[ T_{n^*}^{\text{row}} \left( \frac{N_x N_z N_y}{M_{x,BICC}M_{y,BICC}} \right) \right]. \]

(41)

Recall that \( n^* \) and \( p^* \) corresponded to the \( R_{\text{BICC}} \)-th order statistic of \( \{ T_n \}_{n \in [N]} \), such that \( P \leq n^* \leq P \) and \( 0 \leq p^* \leq P \). To compute \( E[\tau_{\text{Non-h}}] \), we provide effective choice of parameters \( M_{x,BICC}, M_{z,BICC}, \) and \( M_{y,BICC} \), such that they match (21). To do this, we use the following particular choice of parameters: \( M_{x,BICC} = M_{x}, M_{z,BICC} = P M_{z}, \) and \( M_{y,BICC} = M_{y} \). Incorporating these parameters into (41), yields

\[ E[\tau_{\text{BICC}}] = E[T_{n^*}^{\text{comm}}] \left( \frac{N_x N_z}{M_{x}M_{z}} + \frac{N_x N_y}{M_{x}M_{y}} + \frac{N_y N_z}{M_{y}M_{z}} \right) + E\left[ T_{n^*}^{\text{row}} \left( \frac{N_x N_z N_y}{M_{x}M_{y}M_{z}} \right) \right]. \]

(42)

In App. [D] we prove that \( E[\tau_{\text{BICC}}] \leq E[\tau_{\text{Non-h}}] \).

Multilevel coding scheme: To compute the expected finishing time of MLCC which meets constraint (33), we first cluster non-hierarchical coding to three different categories. Each category is characterized by whether \( M_x, M_y, \) or \( M_z \) dominates the partitioning structure of the non-hierarchical scheme.

A non-hierarchical coding scheme of parameters \( (M_x, M_z, M_y) \) is called to be \( M_x \)-dominated if \( M_x \) is the maximum element of the set \( \{M_x, M_z, M_y\} \). Analogous definitions hold for \( M_y \)-dominated and \( M_z \)-dominated. For example, in polynomial codes we have \( M_x = 1 \); hence, polynomial codes are either \( M_x \)- or \( M_y \)-dominated. In contrast, for MatDot codes \( M_y = M_z = 1 \). Therefore, MatDot codes are \( M_z \)-dominated. We now introduce a particular choice of parameters for MLCC in each of above categories. In each category we first specify a subset of parameters with the objective of reducing the input and output communication loads. We then optimize the remaining parameters to minimize the upper and lower bounds of expected finishing time.

Communication load reduction: To reduce the input communication load, it is required to encode data in such a way that many encoded matrices of the set \( \{ A_l(n) \}_{l \in [L]} \) or \( \{ B_l(n) \}_{l \in [L]} \) become equal. As a result, the master is able to amortize the input communication load by sending only a subset of these two sets as representatives of all elements. For instance, if we use a \( M_y \)-dominated scheme as a baseline, we set \( M_{yl} = M_y, M_{zl} = M_z, N_{yl} = N_y, \) and \( N_{zl} = N_z \). These equalize encoded submatrices \( B_l(n) \) across all layers, \( l \in [L] \). Therefore, the master distributes only a single \( \frac{N_y}{M_y} \times \frac{N_z}{M_z} \) encoded submatrix as a representative of the set \( \{ B_l(n) \}_{l \in [L]} \) to each worker. To convey the set \( \{ A_l(n) \}_{l \in [L]} \) to each worker, the master needs to transmit \( \sum_{l=1}^{L} N_{zl} N_{zl} (M_{zl} M_{zl}) = N_{zl} N_{zl} M_{zl} \) real numbers (the proof is provided in App. [E]). The input communication load of MLCC thus satisfies

\[ C_{\text{in}} = \frac{N_y N_z}{M_{x}M_{z}} + \frac{N_y N_z}{M_{y}M_{z}}. \]

(43)

To calculate the output communication load, we incorporate the above choice of parameters into (30). Therefore,

\[ C_{\text{out}} = \frac{N_y N_z}{LM_{x}M_{y}}. \]

(44)

If we use an \( M_y \)-dominated scheme as a baseline, we set \( M_{zl} = M_{x}, M_{zl} = M_z, N_{zl} = N_x, \) and \( N_{zl} = N_z \). With this set of parameters, all encoded submatrices \( A_l(n) \), \( l \in [L] \), assigned to each worker are equal, and we achieve the same input and output communication load as (43) and (44). If we use a \( M_z \)-dominated scheme as a baseline, we set \( M_{zl} = M_{x}, M_{yl} = M_y, N_{zl} = N_z, \) and \( N_{zl} = N_y \). Merging these parameters with (5), (22), and (6) results in \( N_{zl} = N_z M_{zl} / (LM_{z}) \) and \( \sum_{l=1}^{L} N_{zl} = N_z \). Through these results, we again achieve the same input communication load as (43). However, the output communication load is

\[ C_{\text{out}} = \frac{N_y N_z}{M_{x}M_{z}}. \]

(45)

Note that both (44) and (45) satisfy the inequality (33).

Expected finishing time reduction: To obtain \( E[\tau_{\text{MLCC}}] \), we merge the parameter selections made in the previous part into (44) and (26). In either the \( M_x \)- or \( M_y \)-dominated
The above optimization problem is justified in App. F. Given the result of optimization programs yields the optimal set in comparison to computation time. In the following level polynomial code. Algorithm 2 works iteratively layer-by-layer, placing cubes of volume \( v_l = N_{x_l}N_{y_l}K_{sum} \). This yields the needed partitioning of the overall cuboid and avoids overlaps with cuboids placed in previous layers. The cuboid in the \( l \)th layer is then partitioned into \( K_l \) equally sized information blocks according to \( \{ M_{xl}, M_{z_l}, M_{yl} \} \). We comment that rounding errors in Alg. 2 result in at most \( N_xN_z(N_y - \sum_i(N_y/K_{sum} - 1)K_i) = K_{sum}N_xN_z \) extra unit cubes that require additional \( N_xN_zK_{sum} \) basic operations to multiply \( A \) and the last \( K_{sum} \) columns of \( B \). We assign this negligible computation, \( N_xN_zK_{sum} \ll N_xN_zN_y \), to the master.

Algorithm 2: Partition a block \( N_x \times N_z \times N_y \) cuboid into \( L \) task blocks given \( \{ N_{x_l}, N_{z_l}, N_{yl} \} \) and then partition the \( l \)th task block into information blocks given \( \{ K_l, R_l, M_{xl}, M_{z_l}, M_{yl}, K_{sum} \} \), where \( K_{sum} = \sum_i K_i \) and \( K_i = M_{xl}M_{z_l}M_{yl} \).

1. Input: \( L, \{ K_i, R_i, N_{x_l}, N_{z_l}, N_{yl}, M_{xl}, M_{z_l}, M_{yl} \} \) of \( L \), \( K_{sum} \)
2. for \( l \in [L] \):
   1. Slice the \( l \)th task block from the remaining, un-allocated, unit cubes such that the \( l \)th task block contains all basic operations indexed by \( \{(i_x, i_z, i_y) \mid i_x \in [N_x], i_z \in [N_z], i_y \in \left[ \left[ \frac{N_y}{K_{sum}} \right] K_l + \left[ \frac{N_y}{K_{sum}} \right] \sum_{i=1}^{l-1} K_i \right] \} \) 
   2. Given \( \{ M_{xl}, M_{z_l}, M_{yl} \} \), decompose the \( l \)th task block into \( K_l \) equally sized information blocks.
4. end for

VI. EVALUATION

In this section we evaluate the performance of our scheme. We present our results both for numerical simulations and for experiments that were conducted on EC2. We compare BICC and MLCC with non-hierarchical schemes. The latter include uncoded computation, polynomial [9], MatDot [10], polyDot [11], and entangled polynomial [11] coded schemes. These results demonstrate that BICC has the least computation time, and MLCC also outperforms non-hierarchical coding in terms of computation time. On the other hand, decoding time of MLCC can be smaller than that of BICC.

A. Numerical simulations

We first explain the experiment setup for the shifted exponential model. In each trial we generate \( N \) pairs of independent shifted exponential random variables \( (T_n^{comp}, T_n^{comm}) \), \( n \in [N] \), one pair per worker. \( T_n^{comp} \) and \( T_n^{comm} \) are shifted exponential distributions with parameters \( (\mu_{comp}, \alpha_{comp}) \) and \( (\mu_{comm}, \alpha_{comm}) \), respectively. We recall from the discussion in Sec. 1-B that the realization of \( T_n^{comp} \) and \( T_n^{comm} \) set the speed of computation and communication of the \( n \)th worker. Once these two speeds are set, the processor is modeled as progressing through the equal-sized jobs in a (conditionally) deterministic fashion. To optimize the recovery profile of MLCC, we used CVX in MATLAB to solve the convex optimization problem (48).

Effect of \( L \): Figures 6a and 6b plot expected finishing time vs. number of levels \( L \) based on the shifted exponential distribution model corresponding to \( (\mu_{comp}, \alpha_{comp}, \mu_{comm}, \alpha_{comm}) = (10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}) \). In Fig. 6a a polynomial code corresponding to \( N = 300 \) and \( (M_x, M_z, M_y) = (42, 1, 1) \) (an \( M_x \)-dominated scenario) achieves an expected finishing time of 5.98 msec. For the same per-worker computation load \( (K_{sum}/L = 42) \), we plot (the solid line) the performance of MLCC using polynomial coding, for different choices of \( L \). The decrease in finishing time as \( L \) is increased illustrates
that the division of the job into smaller information blocks (larger $L$) results in a reduction in completion time of A*B.
In particular, when compared to non-hierarchical polynomial coding, MLCC observes a 35% improvement in expected finishing time for $L = 96$. We also plot (the dashed dotted line) the expected finishing time of the randomized-order RMLCC (cf. Remark re recovery threshold, equal to $p_{\text{fast}}$. In this case the recovery thresholds are all set equal to $R_{\text{final}}$.

For illustrative reasons, the two points corresponding to bit-interleaved polyDot code and bit-interleaved entangled polynomial code are not shown in Fig. 8. Both of these recovery thresholds are lower than those of the corresponding non-hierarchical schemes.

We now analyze BICC with $L = 8$ using polyDot and entangled polynomial codes with $(M_x, M_y) = (6, 6, 6)$.

**Tradeoffs:** Fig. 8 illustrate the tradeoff between total computation load and the recovery threshold of various non-hierarchical schemes. The total computation load is the maximum number of basic operations that each worker is assigned to complete. This is equal to $C_{\text{comp}}$ in non-hierarchical schemes and $\sum_{l=1}^{L} C_{\text{comp}}/n_{\text{percomp}}$ in MLCC. Both are equal to $N_xN_xN_y/(M_xM_yM_y)$.

In Fig. 8 we first plot (dashed lines) the tradeoff between total computation load and recovery threshold for polyDot and entangled polynomial codes. Entangled polynomial codes outperform polyDot codes at the same recovery threshold. However, entangled polynomial codes have a smaller total computation load than polyDot codes. If the total computation load is fixed, entangled polynomial codes have a lower recovery threshold than polyDot codes. For instance, as is illustrated in Fig. 9 entangled polynomial and polyDot codes with parameter set $(M_x, M_y) = (6, 6, 6)$ have, respectively, recovery thresholds of $R_{\text{ent}} = 221$ and $R_{\text{poly}} = 396$ (cf. (3)). The total computation load of each of these codes is $4.6e6$.

We next plot (solid lines) the tradeoff between the total computation load and the recovery thresholds used across levels for two MLCC approaches. The first is a multilevel entangled polynomial code. The second is a multilevel polyDot code. We apply MLCC with $L = 8$ to entangle polynomial and polyDot codes with $(M_x, M_y) = (6, 6, 6)$. The per-worker computation load in MLCC is at most equal to the computation load of the corresponding non-hierarchical schemes. In other words, workers that complete all their levels have a computation load of $4.6e6$; other workers have lower computation loads. The recovery threshold of the first level, $R_1$, is equal to 41 in the multilevel entangled polynomial code and is equal to 66 in the multilevel polyDot code.

Both of these recovery thresholds are lower than those of the corresponding non-hierarchical schemes: $R_{\text{ent}}$ and $R_{\text{poly}}$, respectively. However, the recovery threshold of the last level, $R_8$, is equal to 864 in the multilevel entangled polynomial code, and is equal to 1377 in the multilevel polyDot code. Both of these recovery thresholds are lower than those of the corresponding non-hierarchical schemes.

We now analyze BICC with $L = 8$ using polyDot and entangled polynomial codes with $(M_x, M_y) = (6, 6, 6)$.

To apply BICC using polyDot and entangled polynomial codes, we use $(M_x, M_y, M_y) = (6, 6, 6)$. This yields BICC with recovery thresholds $R_{\text{B-Pdot}} = 3420$ and $R_{\text{B-Pdot}} = 1775$. Both of these recovery thresholds are approximately $L = 8$ times larger than those of their corresponding non-hierarchical codes. However, the per-subtask computation load of these BICC approaches is 1/8 of their corresponding non-hierarchical codes. For illustrative reasons, the two points corresponded to bit-interleaved polyDot code and bit-interleaved entangled polynomial code are not shown in Fig. 8.

**B. Experiments on Amazon EC2**

From this section onwards we use Python to implement large matrix multiplications on a cluster of $N + 1 \approx \text{“r2.micro” instances} (N \text{ workers and a master})$. We use the function “numpy.dotmul” linked to the library “openBlas” to multiply matrices with entries of type “float32”. We use the package “mpi4py” for the message passing interface between instances.

We first note that to illustrate the strengths and weaknesses of various coded computing schemes on EC2 we need to artificially inject delays into computation (see App. B). In this artificial-straggler scenario we assign workers to be stragglers independently with probability $p$. Workers that are designated stragglers are assigned one more extra computation per-level than non-stragglers (i.e., stragglers are half as fast as non-stragglers). In this artificial-straggler scenario the probability
We plot (solid lines) the serial and parallel decoding times of realizing coded computing on EC2. The decoder we implemented in the master solves a system of linear equations which involves a Vandermonde matrix. Both $A$ and $B$ are $8192 \times 8192$ matrices and the average recovery threshold per level (respectively, $K_{\text{Poly}}, K_{\text{Bi-Poly}}$, and $K_{\text{sum}}/L$, where $P = L$) is set to 8. In Fig. 9a we plot decoding time versus number of levels. We plot (solid lines) the serial and parallel decoding times for MLCC. Each data point on these lines corresponds to a different number of levels, where $K_{\text{Poly}} = K_{\text{BICC}}/P = K_{\text{sum}}/L = 42$, and $N = 300$. (b) where $K_{\text{Poly}} = K_{\text{BICC}}/P = K_{\text{sum}}/L = 4$, and $N = 20$. In both sub-figures $(N_x, N_z, N_y) = (1000, 1000, 1000)$ and $(\mu_{\text{comp}}, \alpha_{\text{comp}}, \mu_{\text{comm}}, \alpha_{\text{comm}}) = (10^{-6}, 10^{-7}, 10^{-8}, 10^{-9})$.

**Fig. 6:** The expected finishing time vs. the number of levels for the shifted exponential distribution (a) where $K_{\text{Poly}} = K_{\text{BICC}}/P = K_{\text{sum}}/L = 42$, and $N = 300$, (b) where $K_{\text{Poly}} = K_{\text{BICC}}/P = K_{\text{sum}}/L = 4$, and $N = 20$. In both sub-figures $(N_x, N_z, N_y) = (1000, 1000, 1000)$ and $(\mu_{\text{comp}}, \alpha_{\text{comp}}, \mu_{\text{comm}}, \alpha_{\text{comm}}) = (10^{-6}, 10^{-7}, 10^{-8}, 10^{-9})$.

**Fig. 7:** Optimal recovery profile of MLCC for different values of $(\mu_{\text{comm}}, \alpha_{\text{comm}})$, where $L = 8$, $N = 20$, $(N_x, N_z, N_y) = (1000, 1000, 1000)$, $K_{\text{Poly}} = K_{\text{BICC}}/P = K_{\text{sum}}/L = 4$ and $(\mu_{\text{comp}}, \alpha_{\text{comp}}, \mu_{\text{comm}}, \alpha_{\text{comm}}) = (10^{-6}, 10^{-7})$.

**Fig. 8:** Total computation load vs. recovery threshold, when $(N_x, N_z, N_y) = (1000, 1000, 1000)$, $M_x = 36$, and $M_y = M_z = 6$. In the hierarchical schemes $L = 8$ and $M_x = M_y = M_z = 6$.

of realizing $q$ stragglers among $N$ workers is ${N \choose q} p^q (1 - p)^{(N - q)}$.

**Effect of $L$ and $P$:** We now discuss the result of implementing coded computing on EC2. The decoder we implemented in the master solves a system of linear equations which involves a Vandermonde matrix. Both $A$ and $B$ are $8192 \times 8192$ matrices and the average recovery threshold per level (respectively, $K_{\text{Poly}}, K_{\text{Bi-Poly}}$, and $K_{\text{sum}}/L$, where $P = L$) is set to 8. In Fig. 9a we plot decoding time versus number of levels. We plot (solid lines) the serial and parallel decoding times for MLCC. Each data point on these lines corresponds to a different number of levels, where $L \in \{1, 2, 4, 8, 16, 32\}$. One can observe that when the decoding process of each level is carried out serially the decoding time of MLCC is very small. It is close to that of polynomial codes. The decoding time of MLCC can be further reduced when the decoding of levels is conducted in parallel. The decoding time of BICC is the largest and increases dramatically as $L$ is increased.

Figure 9b plots the average computation time vs. the number
of levels, where \((N_x, N_z, N_y) = (8192, 8192, 8192)\). We consider the artificial-straggler scenario with \(p = 0.33\). We first plot (dashed line) the average computation time of polynomial coding where \(R_{\text{Pol}} = 8\). In bit-interleaved polynomial coding (and multilevel polynomial coding) we implement a matrix multiplication of dimensions \(N_z \times N_z \times \frac{N_y}{b_0}\) (and \(N_z \times N_z \times \frac{N_y}{b_0}\)) on each of \(N = 12\) workers per subtask (and per level). We assume a \(M_y\)-dominated scheme, where \(M_x = M_z = 1\) and \(M_y = 8\). As in the shifted exponential model, the bit-interleaved polynomial coding (the dotted line) has the smallest average computation time. Compared to polynomial codes, we observe an improvement of 23% in multilevel polynomial coding where \(L = 32\). This improvement increases to 39% in bit-interleaved polynomial codes with \(P = 32\).

**Tradeoffs:** Figure 10 plots a tradeoff between average decoding time and average computation time. For these results, we set \((N_x, N_z, N_y) = (10^4, 10^4, 10^4)\), \(K_{\text{Poly}} = 8, N = 12\) and \(L = P = 10\). The dashed blue line corresponds to hierarchical polynomial coding when we use serial decoding. For the solid red line we decode each level in parallel. These two lines intersect at a small circle marker symbol which corresponds to bit-interleaved polynomial coding with parameters \(P = 10, M_z, \hat{M}_{\text{BICC}} = M_z, \hat{M}_{\text{BICC}} = 1, M_y, \hat{M}_{\text{BICC}} = 80, \) and \(K_{\text{BICC}} = 80\). They meet due to the fact that BICC consists of a single code. Hence, both serial and parallel decoding take the same amount of time, in this case 0.73 sec. The diamond marker symbols on each line corresponds to multilevel polynomial coding with parameters \(L = 10, K_{\text{sum}} = 80, M_{zL} = M_{zL} = 1,\) for all \(l \in [L]\) and \(M_{yL} \in \{12, 12, 11, 10, 9, 7, 5, 2, 0\}\). The triangle marker symbols correspond to hybrid polynomial coding with \(L_{\text{HHCC}} = 5\) levels and \(P_1 = 2\) subtasks per level, \(l \in [L]\). The square marker symbols corresponds to hybrid polynomial coding with \(L_{\text{HHCC}} = 2\) levels and \(P_1 = 5\) subtasks per level. Figure 10 demonstrates that hierarchical coding schemes with a lower number of levels \((L)\) have a less constrained recovery condition, and thus a lower average computation time when compared to hierarchical coding schemes with a larger \(L\). However, the average decoding time in schemes with a larger \(L\) is lower than that of schemes with smaller \(L\), especially when decoding process is conducted in parallel. The average computation time of all hierarchical coding approaches outperforms that of the (non-hierarchical) polynomial coding, which is depicted by the star marker. Polynomial coding has a lower average decoding time when compared to serially decoder in hierarchical coding. However, its decoding time is larger than that of multilevel polynomial coding and hybrid polynomial coding, where \(L_{\text{HHCC}} = 5\) and when decoding is conducted in parallel. Figure 10 also illustrates that, while the uncoded method has zero decoding time, it has the largest average computation time.

**Different size matrices:** In Figure 11a and 11b we plot the sum of the average computation and decoding times for different matrix sizes. To keep the sub-figures comparable we assign the same computation load to each worker. To do this we set \(N_x N_z N_y = 2^{36}\) and \(K_{\text{Poly}} = K_{\text{BICC}}/P = K_{\text{sum}}/L = 8\) in all sub-figures. With this setting, each worker in each scheme is tasked with completing \(2^{36}\) basic operations. In each sub-figure we compare BICC and MLCC with uncoded and polynomial coded schemes. In the hierarchical coding schemes we set \(P = L = 4\). BICC achieves the lowest computation time. Compared to the uncoded scheme, BICC and MLCC, respectively, achieve computation time reductions of, on average, 53% and 47%. Their average reductions are 29% and 20% when compared to polynomial coding.

We now consider decoding time. The uncoded scheme has zero decoding time; no encoding nor decoding occurs. In polynomial coding, the master works with a Vandermonde matrix of dimension \(K_{\text{Poly}} \times K_{\text{Poly}} (8 \times 8)\). The Vandermonde matrices in the decoding phases of BICC and MLCC are, respectively, of dimensions \(K_{\text{B-Poly}} \times K_{\text{B-Poly}} (32 \times 32)\) and
In this paper we introduce hierarchical coded computing to accelerate distributed matrix multiplication. Through our hierarchical design, we can exploit the work completed by stragglers (and by leaders) while, at the same time, providing robustness to stragglers. To apply hierarchical coding to matrix multiplication, we connect the task allocation problem that underlies coded matrix multiplication to a geometric question of cuboid partitioning. We then develop three hierarchical approaches each with its particular strengths and ideal regime of operation. Due to parallelism with coded modulation we term our approaches bit-interleaved coded computation (BICC), multilevel coded computing (MLCC), and hybrid hierarchical coded computing (HHCC). Our proposed schemes allows us to reap significant performance improvement, in terms of computation, decoding, and communication times. We analytically study our scheme under a probabilistic model of computation and communication time. This study is useful in developing design guidelines. Under this model, we numerically show that our method realizes a 66% improvement in the expected finishing time. We also implemented our scheme in Amazon EC2 and measured a 28% improvement in finishing time when compared to state-of-the-art approaches.

A direct extension of this work is to apply hierarchical coding idea into the case of more general computational problem, e.g., tensor multiplication or non-linear computation, rather than just matrix multiplication. Another extension is to use hierarchical coding for a problem of large-scale machine learning algorithm, where matrix multiplication is a building block. Last but not least is to extend our order-statistic analysis to be able to characterize the performance of more general schemes.

REFERENCES

[1] S. Kiani, N. Ferdinand, and S. C. Draper, “Exploitation of stragglers in coded computation,” in IEEE Int. Symp. on Inf. Theory (ISIT), 2018, pp. 1988–1992.

[2] N. Ferdinand and S. C. Draper, “Hierarchical coded computation,” in IEEE Int. Symp. on Inf. Theory (ISIT), 2018, pp. 1620–1624.

[3] S. Kiani, N. Ferdinand, and S. C. Draper, “Hierarchical coded matrix multiplication,” IEEE Canadian Workshop Inf. Theory (CWIT), arXiv preprint:1907.08818, 2019.

[4] ——, “Cuboid partitioning for hierarchical coded matrix multiplication,” IEEE Int. Conf. Machine Learning (Workshop on Coded Machine Learning), arXiv preprint:1907.08819, 2019.

[5] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, “Large scale distributed deep networks,” in Int. Conf. Neural Inf. Proc. Sys. (NIPS), 2012, pp. 1223–1231.

[6] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” IEEE Trans. Inf. Theory, vol. 64, no. 3, pp. 1514–1529, 2018.

[7] S. Dutta, V. Cadambe, and P. Grover, “Short-dot: Computing large linear transforms distributedly using coded short dot products,” in Int. Conf. Neural Inf. Proc. Sys. (NIPS), 2016.

[8] K. Lee, C. Suh, and K. Ramchandran, “High-dimensional coded matrix multiplication,” in IEEE Int. Symp. on Inf. Theory (ISIT), 2017, pp. 2418–2422.

[9] Q. Yu, M. Maddah-Ali, and S. Avestimehr, “Polynomial codes: An optimal design for high-dimensional coded matrix multiplication,” in Int. Conf. Neural Inf. Proc. Sys. (NIPS), 2017.

[10] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, “On the optimal recovery threshold of coded matrix multiplication,” in 2017 55th Annual Allerton Conf. on Communication, Control, and Computing (Allerton), Oct 2017, pp. 1264–1270.

[11] Q. Yu, M. Maddah-Ali, and S. Avestimehr, “Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding,” in IEEE Int. Symp. on Inf. Theory (ISIT), 2018, pp. 1988–1992.
Fig. 11: (a),(b) The average computation plus decoding times of multiplication for matrices of different dimensions; (c),(d) The average finishing times of multiplication for matrices of different dimensions. In all sub-figures $L = P = 4, N = 12, p = 0.33$ and $K_{\text{Poly}} = K_{\text{Poly}}/P = K_{\text{sum}}/L = 8$.

[12] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, “Gradient Coding: Avoiding Stragglers in Distributed Learning,” in Proc. Int. Conf. Machine Learning (ICML), 2017.

[13] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, “A unified coded deep neural network training strategy based on generalized polly dot codes,” in IEEE Int. Symp. on Inf. Theory (ISTI), 2018, pp. 1585–1589.

[14] M. Ye and E. A. Abbe, “Communication-computation efficient gradient coding,” in Int. Conf. on Machine Learning (Int. Machine Learning Society), 2018, p. 9716p.

[15] W. Halbawi, N. Azizan, F. Salehi, and B. Hassibi, “Improving distributed gradient descent using reed-solomon codes,” in IEEE Int. Symp. on Inf. Theory (ISTI), 2018, pp. 2027–2031.

[16] S. Li, S. M. M. Kalan, Q. Yu, M. Soltanolkotabi, and A. S. Avestimehr, “Polynomially coded regression: Optimal straggler mitigation via data encoding,” arXiv preprint:1805.09934, 2018.

[17] S. Dutta, V. Cadambe, and P. Grover, “Coded convolution for parallel and distributed computing within a deadline,” in IEEE Int. Symp. on Inf. Theory (ISTI), 2017, pp. 2403–2407.

[18] N. S. Ferdinand and S. C. Draper, “Anytime coding for distributed computation,” in Annual Allerton Conf. on Communication, Control, and Computing (Allerton), 2016, pp. 954–960.

[19] S. Wang, J. Liu, and N. Shioff, “Coded sparse matrix multiplication,” in Int. Conf. on Machine Learning (ICML), 2018, pp. 5139–5147.

[20] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, “Coded computation over heterogeneous clusters,” IEEE Trans. on Inf. Theory, vol. 65, no. 7, pp. 4227–4242, 2019.

[21] A. Mallick, M. Chaudhari, and G. Joshi, “Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication,” arXiv preprint:1804.10337, 2018.

[22] A. B. Das, L. Tang, and A. Ramamoorthy, “C3les: Codes for coded computation that leverage stragglers,” in IEEE Inf. Theory Workshop (ITW), 2018.

[23] A. Ramamoorthy, L. Tang, and P. O. Vontobel, “Universally decodable matrices for distributed matrix-vector multiplication,” in IEEE Int. Symp. on Inf. Theory (ISTI), 2019, pp. 1777–1781.

[24] M. M. Amiri and D. Giinlutz, “Computation scheduling for distributed machine learning with straggling workers,” in IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP), 2019, pp. 8177–8181.

[25] G. Caire, G. Taricco, and E. Biglieri, “Bit-interleaved coded modulation,” IEEE Trans. on Inf. Theory, vol. 44, no. 3, pp. 927–946, 1998.

[26] U. Wachsmann, R. F. Fischer, and J. B. Huber, “Multilevel codes: Theoretical concepts and practical design rules,” IEEE Trans. on Inf. Theory, vol. 45, no. 5, pp. 1361–1391, 1999.

[27] F. Didier, “Efficient erasure decoding of reed-solomon codes,” in arXiv preprint:0901.1886, 2009.
APPENDIX

A. Statistics observed on Amazon EC2

In this appendix we compare the empirical statistics observed on Amazon EC2 with the shifted exponential model of Sec. V-B. This comparison validates the model we assumed in Sec. V-B.

Communication model: Figure 12a shows the histogram of communication time observed between the master and $N = 12$ workers on Amazon EC2. The master generates two random matrices of dimensions $5000 \times 5000$ for each worker. It then sends each matrix to the respective worker. This communication occurs in a serial manner from the master to all 12 workers. We log the communication times across all workers. We then plot (solid line) the complementary CDF (the CCDF) of master-to-workers communication times derived from the Amazon EC2 empirical data in Fig. 12b. We also plot the CCDF of two shifted exponential distribution. The dotted line corresponds to the shifted exponential distribution that has the same scale and shift parameter to that of the empirical data ($\mu_{\text{comm}} = 0.22$, $\alpha_{\text{comm}} = 0.99$). The dashed line corresponds to the Maximum Likelihood (ML) estimate of the distribution. It can be observed that the shifted exponential distribution is a good match to the empirical communication time data.

Worker computation model: Similarly, the histogram of worker computation times is shown in Fig. 13a. The results are obtained from the time it takes to perform matrix multiplication across $N = 12$ workers over 100 trials. In particular, each worker multiplies two $5000 \times 5000$ random matrices in each trial. Turning to the CCDF lines in Fig. 13b, it can be observed that the CCDF of the experimental computation time (solid line) closely matches the CCDF of the shifted exponential distribution that has the same scale and shift parameters as the empirical data ($\mu_{\text{comm}} = 0.0210$, $\alpha_{\text{comm}} = 3.8037$), plotted using the dotted line. The dashed line plots the CCDF of the shifted exponential distribution obtained from ML estimation. The difference between CCDF of empirical data and the ML estimate is due to the existence of outliers in the empirical data (cf. Fig. 13a).

B. Artificial straggler

In Fig. 14a, we plot the histogram of the artificial-straggler scenario in which we selected a subset of 12 workers to be stragglers according to the Bernoulli(0.33) distribution, i.e., $p = 0.33$. Thus, roughly 33% of workers are going to be stragglers. We plot the CCDF of this scenario in Fig. 14b. Figure 14b illustrates that the ML estimate is a perfect match to the straggle-scenario data.

C. Proof to expected value of order statistics

Given $N$ independent shifted exponential random variables $T_1, \ldots, T_N$ with parameters $\mu$ and $\alpha$, let $T_{R*}$ denote the $R$th order statistics ($T_{[R,N]}$). Then according to (3) the expected value of $T_{R*}$ is

$$E[T_{R*}] = \alpha + \sum_{n=R+1}^{N} \frac{\mu}{n} \approx \alpha + \mu \log \left( \frac{N}{N-R} \right).$$

D. Proof to $\mathbb{E}[T_{BICC}] \leq \mathbb{E}[T_{\text{Non-h}}]$

The $R$th order statistics of the sequence

$$T_{n}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right) + T_{n}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} \right),$$

where $n \in [N]$, is equal to or larger than $R - 1$ elements of the sequence

$$T_{n}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right) + \frac{T_{n}^{\text{comp}} P}{P} \left( \frac{N_x N_y}{M_x M_y} \right),$$

where $n \in [N]$. Let's $i_1, \ldots, i_{R-1}$ be the indices of these $R - 1$ elements. That is, for all $j \in [R-1]$,

$$T_{i_j}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right) + \frac{T_{i_j}^{\text{comp}} P}{P} \left( \frac{N_x N_y}{M_x M_y} \right)$$

is equal to or larger than all $P$ elements

$$T_{j}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right) + \frac{T_{j}^{\text{comp}} P}{P} \left( \frac{N_x N_y}{M_x M_y} \right)$$

where $p \in [P]$. Therefore, the $R$th order statistics

$$T_{R*}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right)$$

is equal or larger than at least $PR$ elements

$$T_{j}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right) + \frac{T_{j}^{\text{comp}} p}{P} \left( \frac{N_x N_y}{M_x M_y} \right)$$

where $j \in [R]$ and $p \in [P]$. Thus, the $R$th order statistics

$$T_{R*}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right)$$

is equal or larger than the $R_{BICC}$th order statistics

$$T_{n}^{\text{comp}} \left( \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} + \frac{N_x N_y}{M_x M_y} \right) + \frac{T_{n}^{\text{comp}} p^*}{P} \left( \frac{N_x N_y}{M_x M_y} \right)$$

where $(n^* - 1)P + p^* = R_{BICC}$. From this, we can conclude that

$$\mathbb{E}[T_{BICC}] \leq \mathbb{E}[T_{\text{Non-h}}].$$

E. Proof to $\sum_{l=1}^{L} N_{z_l} N_{z_l}/(M_{z_l} M_{z_l}) = N_{z_x}/M_{z_x} M_{z_x}$, where $M_{y} = M_{y}, M_{z_l} = M_{z}, N_{y} = N_{y},$ and $N_{z_l} = N_{z_l}$

Recall from (5) that $\sum_{l=1}^{L} N_{z_l} N_{z_l}/N_{y} = N_{z_x}$. Incorporating the above choices of $M_{y}, M_{z_l}, N_{y},$ and $N_{z_l}$ into (5), yields

$$\sum_{l=1}^{L} N_{z_l} = N_{z_x}.$$
that considering (50) and (51) together with the assumption (6) of $M_{ř}$ parameter set $t_{ř}p$ for the shifted exponential models. The latter consists of a shifted exponential model with the parameters and a shifted exponential model with ML estimators. In both sub-figures straggling is natural.

(a) The histogram of master-to-worker communication times, where

\[ \text{Fig. 12: (a)} \] The histogram of computation times, where

As is shown in (32), we also have that $K_{\text{sum}} = \sum_{l=1}^{L} M_{x|l} M_{y|l} = LM_x M_y$. Using the above choice of $M_{y|l}, M_{x|l}, N_{y|l}$, and $N_{x|l}$ in (32), results in

\[ \sum_{l=1}^{L} M_{x|l} = LM_x. \] (51)

Considering (50) and (51) together with the assumption (6) that $N_{x|l} N_{y|l} / (M_{x|l} M_{y|l}) \approx N_x N_y / K_{\text{sum}}$, yields

\[ N_{x|l} = N_x M_{x|l} / (LM_x). \] (52)

Combining these three expressions proves that $\sum_{l=1}^{L} N_{x|l} N_{x|l} / (M_{x|l} M_{x|l}) = N_x N_y / (M_x M_y)$.

F: Derivation of optimization problem (43) and extension to the fast-worker regime

1) Fast-network regime: To obtain the optimization problem (43), we first find the upper and lower bounds for $E[\tau_{\text{MLCC}}]$. Since $E[\tau_{\text{MLCC}}]$ is an expectation of max function (cf. 46) and (47), the upper bound is (28)

\[ E[\tau_{\text{MLCC}}] \leq \left( \max_{t \in [L]} \left[ E \left[ T^{\text{comp}} \right] \right] + \sqrt{ \frac{L-1}{L} \sum_{l=1}^{L} \text{Var} \left[ T^{\text{comp}} \right] } \right) \right) \times \left( \frac{N_x N_y}{LM_x M_y} \right) \right) \] (53)
This upper bound (the right-hand-side of above inequality) can be further bounded and yield the new upper bound

\[
\mathbb{E}[\tau_{\text{MLCC}}] \leq \left( \max_{l \in [L]} \left( \alpha_{\text{comp}} + \mu_{\text{comp}} \log \left( \frac{N}{N - R_l^*} \right) \right) \right) t + \left( \frac{L - 1}{L} \sum_{j=1}^{L} \sum_{i=1}^{N} \frac{1}{i^2} \right) \left( \frac{N_z N_x N_y}{LM_x M_z M_y} \right).
\]

Inequality (54) follows App. C and [29], which result in

\[
\text{Var} \left[ \tau_{\text{comp}} \right] = t^2 \sum_{i=N-R_l+1}^{N} \frac{1}{i^2} \leq t^2 \sum_{i=1}^{N} \frac{1}{i^2}.
\]

Following the convexity property of \( \max \) function, the lower bound of \( \mathbb{E}[\tau_{\text{MLCC}}] \) in a fast-network regime is

\[
\mathbb{E}[\tau_{\text{MLCC}}] \geq \max_{l \in [L]} \left( \mathbb{E}[\tau_{\text{comp}}^*] \right) \left( \frac{N_z N_x N_y}{LM_x M_z M_y} \right) \approx \max_{l \in [L]} \left( \alpha_{\text{comp}} + \mu_{\text{comp}} \log \left( \frac{N}{N - R_l^*} \right) \right) t \times \left( \frac{N_z N_x N_y}{LM_x M_z M_y} \right).
\]

The second approximation in (56) follows App. C.

To optimize the recovery profile, we first use the set \( \{R_l\}_{l \in [L]} \) that minimizes the upper-bound (54). To obtain this optimization problem, we use (54) as the objective function of a minimization problem where \( \{R_l\}_{l \in [L]} \) are the variables. To solve this min-max problems,

\[
\min_{R_l} \left( \max_{l \in [L]} \left( \alpha_{\text{comp}} + \mu_{\text{comp}} \log \left( \frac{N}{N - R_l} \right) \right) t \right) + \left( \frac{L - 1}{L} \sum_{j=1}^{L} \sum_{i=1}^{N} \frac{1}{i^2} \right) \left( \frac{N_z N_x N_y}{LM_x M_z M_y} \right),
\]

we note that the term \( \sqrt{\left( \frac{L - 1}{L} \sum_{j=1}^{L} \sum_{i=1}^{N} \frac{1}{i^2} \right) \left( \frac{N_z N_x N_y}{LM_x M_z M_y} \right)} \) in (57) are independent of \( R_l \), so can be ignored. We then introduce an auxiliary variable \( z \) to recast the problem as the optimization problem (48).

The second constraint, \( R_l \leq R_{l-1} \), is due to the sequential behavior of workers. This means that more workers finish their first subtasks than their second, etc. The third constraint, \( \sum_{l=1}^{L} R_l = LR \), is due to the assumption that the master requires the same total amount of completed computations in MLCC and non-hierarchical coding scheme. We note that while the per-level recovery thresholds are integer, we relax the integer optimization problem into the convex optimization problem (48).

The difference between the upper-bound (54) and the lower-bound (56) is

\[
\sqrt{\left( \frac{L - 1}{L} \sum_{j=1}^{L} \sum_{i=1}^{N} \frac{1}{i^2} \right) \left( \frac{N_z N_x N_y}{LM_x M_z M_y} \right)},
\]

which is constant with respect to the recovery profile \( R_l \). Therefore, the optimization problem which is yielded from minimizing the lower-bound (56) is identical to the optimization problem (48). This shows that the same recovery profile \( \{R_l\}_{l \in [L]} \) minimizes both the upper and lower bounds.

2) Fast-worker regime: Similar to the previous regime, in a fast-worker regime the optimal recovery profile is obtained by minimizing the upper and lower bounds of \( \mathbb{E}[\tau_{\text{MLCC}}] \) in the following optimization problem.

\[
\text{Optimization Problem 2: The solution set to the following}
\]
optimization program yields the set \( \{ R_l = R \}_{l \in [L]} \).

\[
\begin{align*}
\min_{z, \{ R_l \}} & \quad z \\
\text{s.t.} & \quad \left( \alpha_{\text{comm}} + \mu_{\text{comm}} \log \left( \frac{N}{N - R_l} \right) \right) \leq z, \ \forall l \in [L], \\
& \quad R_l \leq R_{l-1} \leq N, \ \forall l \in [L], \\
& \quad \sum_{l=1}^{L} R_l = LR.
\end{align*}
\]  

(58)

**Proof:** Since \( \alpha_{\text{comm}} \) and \( \mu_{\text{comm}} \) are fixed variables that are independent of \( R_l \), we can assume without loss of generality that the objective of above optimization problem is to minimizing \( \log N(N - R_l) \). This is equivalent to minimizing \( R_l \). We can therefore determine optimal recovery profile, by solving the optimization problem

\[
\begin{align*}
\min_{\{ R_l \}_{l \in [L]}} & \quad R_1 \\
\text{s.t.} & \quad \sum_{l=1}^{L} R_l = LR, \ \forall l \in [L], \\
& \quad R_l \leq R_{l-1} \leq N, \ \forall r \in [R], \\
& \quad R_j \in \mathbb{Z}^+, \ \forall j \in [L].
\end{align*}
\]

(59)

Relaxing the integer constraints on the \( \{ R_l \}_{l \in [L]} \), we can reformulate the relaxed problem as the following linear program:

\[
\begin{align*}
\min_{\{ R_l \}_{l \in [L]}} & \quad R_1 \\
\text{s.t.} & \quad \sum_{l=1}^{L} R_l = LR, \ \forall l \in [L], \\
& \quad R_l \leq R_{l-1} \leq N, \ \forall r \in [R].
\end{align*}
\]

(60)

Since \( R_1 \) is the maximum element of the sequence \( \{ R_l \}_{l \in [L]} \), it is always larger or equal than the average, \( \sum_{l=1}^{L} R_l / L \). Therefore, merging this with the first constraint results in \( R_1 \geq R \). The set \( \{ R_l = R \}_{l \in [L]} \) is the only solution that satisfies the above constraints and minimizes the above optimization problem.