An Application of Storage-Optimal MatDot Codes for Coded Matrix Multiplication: Fast k-Nearest Neighbors Estimation

Utsav Sheth[1], Sanghamitra Dutta[1], Malhar Chaudhari[1], Haewon Jeong[1], Yaoqing Yang[1], Jukka Kohonen[2], Teemu Roos[2], Pulkit Grover[1]

***Abstract*—We propose a novel application of coded computing to the problem of the nearest neighbor estimation using MatDot Codes (Fahim et al., Allerton'17) that are known to be optimal for matrix multiplication in terms of recovery threshold under storage constraints. In approximate nearest neighbor algorithms, it is common to construct efficient in-memory indexes to improve query response time. One such strategy is Multiple Random Projection Trees (MRPT), which reduces the set of candidate points over which Euclidean distance calculations are performed. However, this may result in a high memory footprint and possibly paging penalties for large or high-dimensional data. Here we propose two techniques to parallelize MRPT that exploit data and model parallelism respectively by dividing both the data storage and the computation efforts among different nodes in a distributed computing cluster. This is especially critical when a single compute node cannot hold the complete dataset in memory. We also propose a novel coded computation strategy based on MatDot codes for the model-parallel architecture that, in a straggler-prone environment, achieves the storage-optimal recovery threshold, *i.e.*, the number of nodes that are required to serve a query. We experimentally demonstrate that, in the absence of straggling, our distributed approaches require less query time than execution on a single processing node, providing near-linear speedups with respect to the number of worker nodes. Our experiments on real systems with simulated straggling, we also show that in a straggler-prone environment, our strategy achieves a faster query execution than the uncoded strategy.**

***Index Terms*—MRPT, MatDot codes, K-NN search, straggler and failure tolerance**

I. INTRODUCTION

We consider the problem of finding the *k* nearest neighbors of a query point in a given high-dimensional dataset. To solve this problem efficiently, our goal is to speed up an existing algorithm [1] by parallelizing it, and to make it resilient to *stragglers*. The *k*-nearest neighbor (*k*-NN) problem is often a first step used in a variety of real world applications (see [1]) including genomics, personalized search, network security, and web based recommendation systems.

In the era of Big Data, *k*-NN algorithms are often a bottleneck, as data and dimensionalities grow [2]. There is a rich body of work on fast nearest neighbor retrieval. The existing work can be broadly classified into three categories. The first category of methods speeds up *k*-NN retrieval by reducing the space over which exact distance calculations are performed, by employing space partitioning data structures [3]–[5]. The second category of techniques improves retrieval times by system level parallelism [6], [7]. Algorithms in the first and second categories are seldom scalable as they require that the whole dataset be held in (shared) memory for optimal performance. For large high-dimensional datasets, marshaling enough resources on a single system is challenging. A third category of methods aims to overcome this challenge by parallelizing storage and computation in a distributed setting. PANDA [8] and DSI sharding [9] are examples of data parallel implementations of distributed *k*-NN algorithms. While these techniques rely on specialized or high performance hardware (e.g. Edison supercomputer for PANDA), the general trend in the distributed systems has been to use general purpose commodity systems [10] [11]. These approaches also do not address a more serious issue — the effects of node failures and slow nodes, or "stragglers". Schroeder and Gibson [12] observed as many as 1,159 system failures per year at the Los Alamos National Laboratory. Dean et al. [13] study a real Google service and observe that the slowest 5% of requests are responsible for half of the total 99th percentile latency.

Recently, the idea of *Coded Computing* [14]–[22] has been found to be very useful in combating stragglers and faults, by the efficient use of novel erasure-codes to create redundancy in computing. In this paper, we use one such coded computing technique called *MatDot codes* to speed up an approximate k-NN algorithm called *Multiple Random Projection Trees* (MRPT) [1], that achieves the storage-optimal recovery threshold[1] of $2m-1$. We note that for matrix multiplication $\mathbf{X}^T\mathbf{Q}$ or matrix-vector product $\mathbf{X}^T\mathbf{q}$ under the constraint that only $\frac{1}{m}$ fraction of each operand can be stored at each node, MatDot codes use vertical block-partitioning of the first matrix $\mathbf{X}^T$ as compared to other existing strategies that use either horizontal partition or a combination of both (see [17], [23], [24]); and it turns out that vertical partitioning of the first matrix is better suited for the coded MRPT problem formulation.

This coded computing problem ensues from our broader goal in this work, which is to speed up MRPT by employing

This work was supported by the NSF CNS-1702694, and the Academy of Finland under the WiFIUS program, the Academy of Finland COIN CoE and NSF CCF 1350314.

[1]Carnegie Mellon University, pulkit@cmu.edu

[2]University of Helsinki, teemu.roos@cs.helsinki.fi

[1]Number of workers required to wait for out of the total in the worst case

data and model parallelism and straggler-tolerant computing techniques. We differentiate here between *data parallelism* and *model parallelism*. In data parallelism, different nodes process different pieces of data, but each node performs all the computations relevant to the entire model. In model parallelism, the model itself is parallelized across nodes.

MRPT partitions the search space to retrieve approximate k nearest neighbors of the query $\mathbf{q}$. It uses a combination of random projection trees and voting to achieve fast queries and high accuracy. In this paper, we propose two enhancements to the MRPT algorithm by parallelizing it in a distributed setting. Our contributions are as follows:

- We propose a distributed implementation of MRPT exploiting data parallelism that experimentally demonstrates faster queries than a single node implementation, even when using CPUs with lower clock speeds. Additionally, the cloud based virtual machines we use in our experiments for parallel MRPT have a non-zero steal time, *i.e.*, they may be required to wait while others are being served.
- We formulate a coded computing problem for MRPT, and then apply coded matrix multiplication strategies, namely MatDot and Systematic MatDot codes to further reduce the query time for the model parallel architecture in a system that is prone to straggling.

The rest of the paper is organized as follows. In Section II we explain the MRPT algorithm and describe how it reduces the search-space through projections and voting. In Section III we introduce the Data Parallel Model Implementation of MRPT. In Section IV we introduce our proposed model parallel implementation of MRPT and then describe the application of MatDot codes and systematic MatDot codes in our model parallel architecture to achieve a lower recovery threshold [23] under straggling. The model parallel architecture is ideal for applications where system components are unreliable and accuracy is important. In Section V we experimentally demonstrate the advantages of our approach. A conclusion is provided in Section VI.

II. Preliminaries

A. MRPT Algorithm

This section briefly describes the two stages of the MRPT algorithm: (i) off-line index construction stage and (ii) online query stage. Assume that we are given a d-dimensional dataset $\mathcal{X}$ consisting of N points, represented as a $d \times N$ matrix $\mathbf{X}$. Given a query point $\mathbf{q}$, the problem of k-nearest neighbors involves finding a set of points $\kappa \subseteq \mathcal{X}$ such that $|\kappa| = k$ and $dist(\mathbf{x}, \mathbf{q}) \leq dist(\mathbf{y}, \mathbf{q})$ for each $\mathbf{x} \in \kappa$, $\mathbf{y} \in \mathcal{X} \backslash \kappa$, and the function $dist(\cdot)$ is the distance function in the d-dimensional Euclidean space given by:

$$dist(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\| = \sqrt{\|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\mathbf{u} \cdot \mathbf{v}} \quad (1)$$

where $\mathbf{u}$ and $\mathbf{v}$ are two vectors in this space.

In the MRPT algorithm, a sparse d-dimensional random projection vector $\mathbf{r}$ is chosen, in which each entry r_i is sampled from the following distribution:

$$r_i = \begin{cases} \mathcal{N}(0,1) & \text{with probability } a \\ 0 & \text{with probability } 1 - a. \end{cases}$$

Typically, the sparsity parameter a can be chosen as $\frac{1}{\sqrt{d}}$, as in [1], to obtain good accuracy. Now, each d-dimensional data-point $p \in \mathcal{X}$ is then projected onto the sparse vector $\mathbf{r}$. The dataset $\mathcal{X}$ is then divided into two subsets at the median point of the projected values. The process is then repeated recursively for every subset at a level, with a new random vector $\mathbf{r}$ chosen for that tree level, until depth ℓ is reached. Thus, for every tree $t \in \mathcal{T}$, the entire dataset $\mathcal{X}$ is partitioned into 2^ℓ cells (or leaves), denoted as $L_1, L_2, \ldots, L_{2^l}$, all of which contain $\lceil \frac{N}{2^\ell} \rceil$ or $\lfloor \frac{N}{2^\ell} \rfloor$ data-points.

Online Query Stage in MRPT: Given a d-dimensional query vector $\mathbf{q}$, the first step in the MRPT query stage is to generate a candidate set of indices (pruned data-point indices) $S \subset \{1, 2, \ldots, N\}$ such that $|S| \ll N$.

For each Random Projection (RP) tree $t \in \mathcal{T}$, at each level the query vector $\mathbf{q}$ is projected onto the random vector $\mathbf{r}$ for that level and then assigned a branch based on whether its value is greater than or less than the median of the projections of all other data-points with $\mathbf{r}$. This process is then repeated recursively until a leaf is reached.

Each tree had already partitioned the dataset $\mathcal{X}$ into 2^l cells or leaves. For $1 \leq t \leq T$, let $f_t(\cdot)$ be defined as:

$$f_t(\mathbf{x} : \mathbf{q}) = \sum_{i=1}^{2^\ell} \mathbb{1}(\mathbf{x} \in L_i, \mathbf{q} \in L_i) \quad (2)$$

where $\mathbb{1}(\mathbf{x} \in L_i, \mathbf{q} \in L_i)$ denotes the indicator function that returns 1 if both $\mathbf{x}$ and $\mathbf{q}$ reside in the same cell. Let $F(\cdot)$ be a function that returns the number of trees in which $\mathbf{x}$ and $\mathbf{q}$ occur in the same leaf, defined as follows:

$$F(\mathbf{x}; \mathbf{q}) = \sum_{t=1}^{T} f_t(\mathbf{x}, \mathbf{q}). \quad (3)$$

The candidate set of indices (pruned points) S can then be finally chosen as follows:

$$S = \{j \subset \{1, 2, \ldots, N\} : \mathbf{x}_j \in \mathcal{X} \text{ and } F(\mathbf{x}_j; \mathbf{q}) \geq \nu\} \quad (4)$$

Here, ν is a pre-configured parameter known as the *voting threshold*. Thus, the set S denotes the set of indices $\subset \{1, 2, \ldots, N\}$ for which at least ν trees have found the corresponding data-point $\mathbf{x}_j$ in the same cell as $\mathbf{q}$.

Finally, exact distance calculations are performed for each $\mathbf{x}_j$ with $j \in S$, to obtain the approximate k nearest neighbors to the query-vector $\mathbf{q}$. The algorithm for this stage is mentioned in Algorithm 1. Here, $\texttt{TREE_QUERY}(\mathbf{q}, t)$ is a function corresponding to tree t that returns the pruned collection of the indices of the data-points that lie in the same cell (or leaf) as $\mathbf{q}$. Thus, $\texttt{TREE_QUERY}(\mathbf{q}, t)$ gives $\{j \subset \{1, \ldots, N\} : \mathbb{1}(\mathbf{x}_j \in L_i, \mathbf{q} \in L_i) = 1$ for some $L_i\}$. The exact distance calculation is discussed again in Section IV.

B. Coded Computing: Coded Matrix Multiplication

Coded computing combines distributed numerical algorithms and error correcting codes (ECCs) to mitigate unreliable processors and randomness in their response time. In

Algorithm 1 The MRPT Query Phase

```
1: procedure APPROXIMATE_KNN(q, k, T, ν)
2:     S ← ∅
3:     Let votes = [0, ..., 0] be a new n-dimensional array
4:     for t in T do
5:         for point in TREE_QUERY(q, t) do
6:             votes[point] ← votes[point] + 1
7:             if votes[point] = ν then
8:                 S ← S ∪ {point}
9:     return EXACT_KNN(q, k, S)
```

this work, we focus on coded matrix multiplication as matrix multiplication is the main bottleneck in MRPT algorithm.

System Model: We want to compute $\boldsymbol{C} = \boldsymbol{AB}$ where $\boldsymbol{A}$ and $\boldsymbol{B}$ are N-by-N matrices. *A master node* distributes the computation to P *worker nodes*. A worker node has limited memory/computing power, so each node can receive the $1/m$-th fraction of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. After completing its computation, a worker reports the result to *a fusion node. Recovery threshold* is defined as the worst-case number of workers needed to recover the final result.

Let K be the number of workers needed to complete the computation if all worker nodes are reliable (K is different depending on how we split the matrix). In reality, some processors are significantly slower than the others due to queuing delays or random faults in the processor [14]. Without any reliability measure to alleviate straggler problems, computation completion time would be dominated by few stragglers. Our aim is to use more than K worker nodes by adding some redundancies, which are carefully designed by applying the ideas from coding theory, so that the whole computation can be resilient to stragglers.

MatDot codes: The matrix $\mathbf{A}$ is split vertically into m column blocks, and $\mathbf{B}$ is split horizontally into m row blocks:

$$\mathbf{A} = [\mathbf{A}_1 \ \mathbf{A}_2 \ \dots \ \mathbf{A}_m], \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \vdots \\ \mathbf{B}_m \end{bmatrix}, \tag{5}$$

where $\mathbf{A}_i, \mathbf{B}_i$ $(i = 1, \dots, m)$ are $N \times N/m$ and $N/m \times N$ dimensional submatrices, respectively.

The matrices $\mathbf{A}$ and $\mathbf{B}$ are then encoded as polynomials:

$$p_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_i x^{i-1}, \quad p_{\mathbf{B}}(x) = \sum_{j=1}^{m} \mathbf{B}_j x^{m-j}. \tag{6}$$

A master node distributes encoded matrices, $p_{\mathbf{A}}(\alpha_i)$ and $p_{\mathbf{B}}(\alpha_i)$ to the i-th worker node ($i = 1, \dots, P$). Then the i-th worker node computes the following product at $x = \alpha_i$:

$$p_{\mathbf{C}}(x) = \sum_{i=1}^{m}\sum_{j=1}^{m} \mathbf{A}_i \mathbf{B}_j x^{m-1+(i-j)}, \tag{7}$$

and returns the result to the master node. Note that the coefficient of x^{m-1} in $p_{\mathbf{C}}(x)$ is $\mathbf{C} = \sum_{i=1}^{m} A_i B_i$. Since $p_{\mathbf{C}}(x)$ is a polynomial of degree $2m-2$, its coefficients can be recovered by the master node as soon as it receives the values of $p_{\mathbf{C}}(x)$ at any $2m-1$ distinct points. Hence the recovery threshold is $K = 2m-1$. This is provably the optimal recovery threshold, when a worker node can store $\frac{1}{m}$-th fraction of each input matrix [23].

Systematic MatDot codes: A code is called *systematic* if, for the first m worker nodes, the output of the r-th worker node is the product $\mathbf{A}_r\mathbf{B}_r$. We refer to the first m worker nodes as *systematic worker nodes*. Having systematic nodes is useful because if all the systematic nodes complete their computation in time, there is no need for decoding. Systematic MatDot codes are achieved by applying different encoding polynomials. Let $p_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_i L_i(x)$ and $p_{\mathbf{B}}(x) = \sum_{i=1}^{m} \mathbf{B}_i L_i(x)$ where $L_i(x)$ is defined as follows for $i \in \{1, \dots, m\}$:

$$L_i(x) = \prod_{j \in \{1,\dots,m\}\setminus\{i\}} \frac{x - x_j}{x_i - x_j}. \tag{8}$$

Using these polynomials, the worst-case recovery threshold remains the same as non-systematic MatDot codes [23].

III. DATA PARALLEL MRPT

Now, we model MRPT as a problem in data parallelism and describe our first strategy to parallelize the algorithm. Consider a distributed computing cluster having a single *master node* and P *worker nodes* as shown in Fig. 1.

Given a $d \times N$ matrix $\mathbf{X}$ representing the set of data-points $\mathcal{X}$ and a cluster with P worker nodes, we randomly split $\mathbf{X}$ vertically into P disjoint, vertical partitions $\mathbf{X}_i$ for $i \in \{1, 2, \dots, P\}$.Thus, $\mathbf{X} = \left[\mathbf{X}_1 | \mathbf{X}_2 | \dots | \mathbf{X}_P\right]$, where each $\mathbf{X}_i$ is of dimension $d \times \frac{N}{P}$.

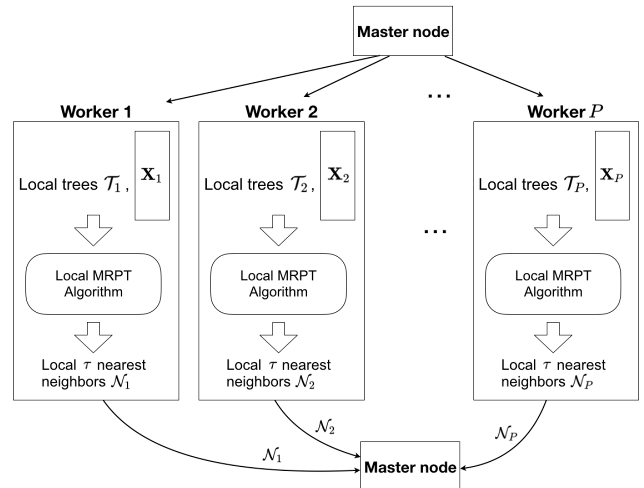


Fig. 1: Data Parallel MRPT Architecture.

We distribute each partition $\mathbf{X}_i$ across the P worker nodes in the cluster such that the worker W_i contains partition $\mathbf{X}_i$. Each worker then runs the MRPT index construction algorithm described in Section II-A to build a local MRPT set of trees $\mathcal{T}_i$ from its partition $\mathbf{X}_i$ of $\mathbf{X}$.

Given a query $\mathbf{q}$, the master node transmits $\mathbf{q}$ to each worker node W_i. Then W_i uses the MRPT query algorithm described in Section II-A on its trees $\mathcal{T}_i$ to determine the τ

nearest neighbors of $\mathbf{q}$. We use the same voting threshold ν in all workers. After local voting, a local exact distance calculation step is performed to narrow down to τ data-points. Finally, each worker node W_i then returns the indices of its set of τ $(\geq k)$ nearest neighbors N_i, from the partition $\mathbf{X}_i$ of $\mathbf{X}$, to the master node along with their exact distances from the query $\mathbf{q}$. Then, the master node determines the final set of k nearest neighbors to $\mathbf{q}$ from all the $P\tau$ candidate data-point indices received from all the worker nodes, *i.e.*, the indices of all the data-points in the set $\cup_{i=1}^{P} N_i$, by sorting the data-points based on their exact distances.

Remark (1) *: Recall that $\kappa \subseteq \mathcal{X}$ (with $|\kappa| = k$) is the set of the true k nearest neighbors of $\mathbf{q}$. We let $\kappa_i \subseteq \kappa$ be the set of the true nearest neighbors to $\mathbf{q}$ that lie in worker W_i. To achieve maximum accuracy, we must have $\kappa_i \subseteq N_i$ and therefore the value of ν chosen for the system must be such that $|\kappa_i|$ is much less than τ. In fact, a higher value of τ implies higher chance of containing the desired set κ_i, though it comes with increased communication cost from worker to master and increased computation at the master node.*

IV. MODEL PARALLEL MRPT

In this section we discuss a model parallel architecture for approximate k nearest neighbor search using MRPT. We then propose two enhancements to the model parallel architecture that apply coded distributed matrix multiplication techniques that achieve the optimal recovery threshold in a system that is prone to straggling.

A. Problem Formulation for Model Parallel MRPT

Consider the d-dimensional dataset $\mathcal{X}$ as before. In the model parallel architecture, given a query $\mathbf{q}$, we first find the possible candidate set of indices (pruned indices) S using the recursive algorithm described in Section II-A. Now the search space for the true nearest neighbors κ reduces to the set of data-points whose indices are in S, *i.e.*, $\kappa \subseteq \{\mathbf{x}_j : j \in S\}$.

To find the set κ, we compute the exact Euclidean distance from each data-point $\mathbf{x}_j$ (for $j \in S$) to the query point $\mathbf{q}$. Examining the terms constituting the Euclidean distance in (1), the Euclidean norm $\|\mathbf{x}_j\|$ for each $\mathbf{x}_j \in \mathcal{X}$ can be precomputed and the same can be done to get $\|\mathbf{q}\|$. We must now only compute the dot product $\mathbf{x}_j \cdot \mathbf{q}$ to obtain the Euclidean distances from each $\mathbf{x}_j$ to $\mathbf{q}$.

To do this, we first represent the data-points indexed in the set S as a $d \times |S|$ matrix $\mathbf{X}(S)$ that contains only the data-points $\mathbf{x}_j$ (columns of $\mathbf{X}$) such that $j \in S$. The transpose of this matrix is the $|S| \times d$ matrix $\mathbf{X}(S)^T$ that essentially denotes all the rows of the matrix $\mathbf{X}^T$ indexed in S.

Consider the column vector $\mathbf{w}$ such that $\mathbf{w} = \mathbf{X}(S)^T\mathbf{q}$. Note that each element of the vector $\mathbf{w}$ corresponds to the dot-product $\mathbf{x}_j \cdot \mathbf{q} = \mathbf{x}_j^T\mathbf{q}$ for some $j \in S$. The Euclidean distance from $\mathbf{x}_j$ to $\mathbf{q}$ can now be determined as all the terms in (1) are known to us, which includes the individual norms as well as the dot product $\mathbf{x}_j \cdot \mathbf{q}$. The problem thus reduces to the following: compute the vector $\mathbf{w} = \mathbf{X}(S)^T\mathbf{q}$ in a distributed computing cluster where $\mathbf{X}^T$ is known in advance. Since the computation $\mathbf{w} = \mathbf{X}(S)^T\mathbf{q}$ is the only stage of the algorithm that must be done at runtime (in the online stage) and scales linearly with d, we now discuss several scalable strategies that compute vector $\mathbf{w}$ in a distributed setting.

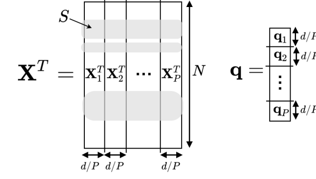


Fig. 2: Partitioning the data matrix $\mathbf{X}$ and the query vector $\mathbf{q}$ in Model Parallel MRPT.

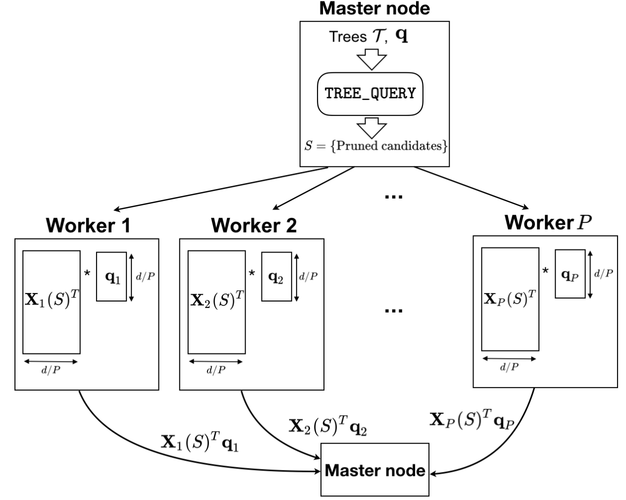


Fig. 3: Model Parallel Architecture with Uncoded Distributed Matrix Multiplication.

B. Uncoded Distributed Matrix-Vector Multiplication

We split $\mathbf{X}^T$ into P equal partitions as follows (see Fig. 2):

$$\mathbf{X}^T = \begin{bmatrix} \mathbf{X}_1^T & \mathbf{X}_2^T & \dots & \mathbf{X}_P^T \end{bmatrix}. \tag{9}$$

Now consider a cluster consisting of one *master node* and P *worker nodes* as shown in Fig. 3. Each partition $\mathbf{X}_i^T$ is distributed across the worker nodes such that worker W_i stores the partition $\mathbf{X}_i{}^T$ in advance (off-line).

Note that, if $\mathbf{X}^T$ is partitioned using the strategy just discussed, the matrix $\mathbf{X}(S)^T$ also gets partitioned as follows:

$$\mathbf{X}(S)^T = \begin{bmatrix} \mathbf{X}_1(S)^T & \mathbf{X}_2(S)^T & \dots & \mathbf{X}_P(S)^T \end{bmatrix} \tag{10}$$

In the online phase, we only split the query $\mathbf{q}$ into P equal partitions, $\{\mathbf{q}_i : i \in \{1, \dots, P\}\}$ (again see Fig. 2). The product $\mathbf{w}$ can then be expressed as:

$$\mathbf{w} = \sum_{i=1}^{P} \mathbf{X}_i(S)^T\mathbf{q}_i \tag{11}$$

Given a query $\mathbf{q}$ for which the k nearest neighbors must be determined, the master node first computes the possible candidate set S for $\mathbf{q}$ from its MRPT index set of trees $\mathcal{T}$ and then transmits the set S and partition $\mathbf{q}_i$ of $\mathbf{q}$ to worker

node W_i. For every S, each worker node W_i only fetches the matrix $\mathbf{X}_i(S)^T$ from $\mathbf{X}_i^T$ already stored in its memory. It then computes the product $\mathbf{X}_i(S)^T\mathbf{q}_i$ and returns the resulting vector to the master node. The master node can thus compute the vector $\mathbf{w}$ by adding the results using (11), and determine the k nearest neighbors using the exact distances.

C. Coded Distributed Matrix-Vector Multiplication using MatDot Codes

In order to successfully compute the vector $\mathbf{w}$ using the strategy described in Section IV-B, the master node must wait for every worker node W_i to successfully return the product $\mathbf{X}_i(S)^T\mathbf{q}_i$. In a straggler-prone environment, this might cause unprecedented delays in computation. Thus, to avoid waiting for all nodes and be able to recover the matrix-vector product by only waiting for some out of all workers to finish, we will now apply the MatDot-based distributed matrix multiplication strategy [23].

We partition the matrix $\mathbf{X}^T$ vertically again, but into m partitions instead of P as follows:

$$\mathbf{X}^T = \begin{bmatrix}\mathbf{X}_1^T & \mathbf{X}_2^T & \dots & \mathbf{X}_m^T.\end{bmatrix} \tag{12}$$

We then use the following encoding polynomial:

$$P_{\mathbf{X}^T}(\beta) = \sum_{j=1}^{m} \mathbf{X}_j^T \beta^{j-1}. \tag{13}$$

The rows of $P_{\mathbf{X}^T}(\beta)$ indexed in set S actually represent the following polynomial:

$$P_{\mathbf{X}^T(S)}(\beta) = \sum_{j=1}^{m} \mathbf{X}_j(S)^T \beta^{j-1}. \tag{14}$$

We will be referring to this observation later.

Now, given a cluster with a master node and P worker nodes, as shown in Fig. 4, each worker node W_i is initialized with a different β_i, using which it computes the polynomial $P_{\mathbf{X}^T}(\beta_i)$ in (13). This encoding step can be performed off-line as $\mathbf{X}^T$ is known in advance.

During the online stage, given a query $\mathbf{q}$ for which the k nearest neighbors must be determined, the master node first partitions $\mathbf{q}$ into m parts: $\{\mathbf{q}_j : j \in \{1, 2, \dots, m\}\}$. We then use the following encoding polynomial:

$$P_{\mathbf{q}}(\beta) = \sum_{j=1}^{m} \mathbf{q}_j \beta^{m-j} \tag{15}$$

As in Section IV-B, the master node first determines the candidate set S. It then transmits S and the encoded query $P_{\mathbf{q}}(\beta_i)$ obtained from (15) to worker W_i. The worker W_i then fetches only the matrix $P_{\mathbf{X}(S)^T}(\beta_i)$ from its stored $P_{\mathbf{X}^T}(\beta_i)$ (recall (14)) which essentially denotes all the rows of $P_{\mathbf{X}^T}(\beta_i)$ indexed in S. Then, it computes the product $P_{\mathbf{X}(S)^T}(\beta_i)P_{\mathbf{q}}(\beta_i)$ and returns the result to the master node.

The coefficient of β^{m-1} in the polynomial $P_{\mathbf{X}(S)^T}(\beta)P_{\mathbf{q}}(\beta)$ turns out to be our desired desired matrix-vector product $\mathbf{X}(S)^T\mathbf{q} = \sum_{j=1}^{m} \mathbf{X}_j(S)^T\mathbf{q}_j$ from the property of MatDot codes. We need to evaluate the polynomial at only $2m-1$ distinct points so as to determine

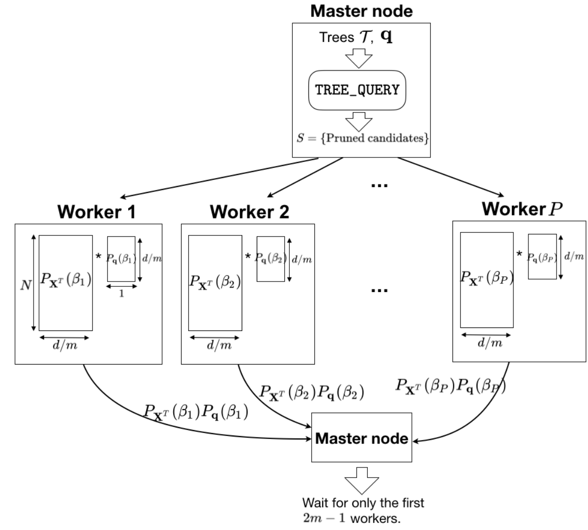


Fig. 4: Model Parallel Architecture with MatDot Codes.

the coefficient for every power of β. The master node must therefore wait for at least $2m-1$ worker nodes following which it can determine the term $\sum_{i=1}^{m} \mathbf{X}_i(S)^T\mathbf{q}_i$ using polynomial interpolation. We then follow the strategy of comparing the exact distances in Section IV-A to obtain the set of the k nearest neighbors to $\mathbf{q}$.

D. Coded Distributed Matrix-Vector Multiplication using Systematic MatDot Codes

In our prior work [23] where we proposed MatDot Codes, we also introduced their systematic variant. Their advantage is that, while for MatDot Codes the recovery threshold is always $2m-1$, for systematic MatDot Codes one might sometimes only need m nodes to finish, although $2m-1$ is the worst-case value. In this section, we apply the systematic MatDot code to the MRPT problem.

Similar to the previous case, we first partition $\mathbf{X}^T$ vertically into m partitions, but then use a different encoding function:

$$P_{\mathbf{X}^T}(\beta) = \sum_{j=1}^{m} \mathbf{X}_j^T L_j(\beta), \tag{16}$$

where $L_j(\beta)$ is given by:

$$L_j(\beta) = \prod_{r\in\{1,2,\dots,m\}\setminus j} \frac{\beta-\beta_r}{\beta_j-\beta_r} \tag{17}$$

Consider a cluster consisting of one master node and P worker nodes. Worker W_i is assigned a value β_i using which it computes the polynomial in (16). This encoding is performed in advance, in the off-line stage. Interestingly, the workers $\{W_i : i = 1, 2, \dots, m\}$ turn out to be the *systematic* worker nodes, which contain uncoded partitions of $\mathbf{X}^T$.

In the online phase, given a query $\mathbf{q}$, the master node first partitions $\mathbf{q}$ into m partitions and then uses the following encoding function:

$$P_{\mathbf{q}}(\beta) = \sum_{j=1}^{m} \mathbf{q}_j L_j(\beta) \tag{18}$$

where $L_j(\beta)$ is given by (17). The master node then transmits candidate set S and $P_{\mathbf{q}}(\beta_i)$ to each worker node. Worker W_i is responsible for computing the product $P_{\mathbf{X}(S)^T}(\beta_i)P_{\mathbf{q}}(\beta_i)$. We first consider the case when the first m workers to successfully complete their computation are the *systematic* worker nodes. We can then obtain the vector $\mathbf{w}$ as follows:

$$\mathbf{w} = \sum_{i=1}^{m} P_{\mathbf{X}(S)^T}(\beta_i)P_{\mathbf{q}}(\beta_i) \tag{19}$$

If the results of the first $2m-1$ successful workers do not contain results from the m systematic nodes, then the master interpolates the polynomial $P_{\mathbf{X}(S)^T}(\beta)P_{\mathbf{q}}(\beta)$. It then computes this polynomial product at each $\beta_i \in \{\beta_1,\beta_2,\ldots,\beta_m\}$. Finally, it computes the vector $\mathbf{w}$ using (19). Note that in the ideal case, *i.e.*, when all the systematic worker nodes finish first, we only needed m nodes to finish as opposed to the worst-case recovery threshold of $2m-1$. We can now proceed with the steps of comparing exact distances (see Section IV-A) to retrieve the *k* nearest neighbors of $\mathbf{q}$.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of data and model parallel MRPT in terms of both accuracy and speed. All of our experiments were conducted on Amazon Elastic Compute Cloud instances [25].

The STL-10 dataset [26] is a dataset of $N = 100000$ images each of dimension $d = 9216$ used in unsupervised image classification algorithms, while GIST [27] is a popular dataset with $N = 1000000$ and $d = 960$ used in ANN algorithms. These datasets provide us with a good mix of dimensionality and number of datapoints to evaluate our proposed strategies. The MRPT parameters used for the experiments are provided in Table I.

TABLE I: MRPT Parameters

Dataset	ℓ	ν	*Number of Trees*	*Projection Sparsity*
STL-10	7	25	900	0.01
GIST	9	10	900	0.032

We evaluate the accuracy of our implementation using recall defined as: $\dfrac{|\kappa_{MRPT} \cap \kappa|}{k}$, where $\mathbf{q}$ is the query whose true *k* nearest neighbors is the set κ and κ_{MRPT} is the set of *k* nearest neighbors returned by the algorithm.

For the single node MRPT baseline, we used a compute optimized *c5.large* instance with two 3GHz Intel Xeon Platinum processors and 4 GB of memory. For the data parallel and model parallel architectures, we used *t2.medium* instances with two 2.3 GHz Intel Broadwell processors and 4 GB of memory. All instances are provisioned with 40GB HDD secondary storage. Note that the hardware used for our baseline experiments is superior to that used to evaluate our parallel strategies. In all experiments we ran 500 queries sequentially with $k = 10$. We consider the average result of 50 runs for each experiment. The experiments were conducted in a cluster consisting of 1 master node and 16 worker nodes.

In the experiments with MatDot codes and systematic MatDot codes, we used encoding polynomials of degree 2.

A. System Constraints

All our experiments are conducted on systems with limited memory. Thus the MRPT algorithm has to use the disk to hold the index $\mathcal{T}$ and the actual data points. In comparison, the data parallel and model parallel architectures use less memory and avoid disk penalties. In the data parallel architecture, we reduce the amount of points each worker must hold by a factor of *P*. Additionally, the MRPT indexes become smaller as each worker holds a smaller fraction of the dataset. In the model parallel architecture, the master node can discard the components for each point after tree construction; it only needs the index. At each worker node, the memory requirement is at least halved.

B. Simulating Stragglers on Amazon Web Services

To demonstrate the effects of stragglers in the model parallel architecture, we sample the minimum time T_i a worker W_i must take to complete a matrix multiplication for a query from the shifted Exponential [14], [16] and Weibull distributions [28], [29] shown in (20) and (21) respectively.

$$Pr[T_i \le t] = 1 - e^{-\frac{\mu_i}{l_i}(t - a_i l_i)} \tag{20}$$

$$Pr[T_i \le t] = 1 - e^{\left(-\frac{\mu_i}{l_i}(t - a_i l_i)\right)^{\alpha_i}} \tag{21}$$

Here, l_i is the number of row vectors loaded at worker W_i for matrix multiplication, $a_i > 0$ is the shift parameter, $\alpha_i > 0$ is the shape parameter for the Weibull distribution, $\mu_i > 0$ is the straggling parameter for W_i, and $t \ge a_i l_i$. For our experiments, we set each μ_i, a_i, and α_i to some constants μ, a, and α to maintain homogeneity in minimum computation times across workers. The parameters chosen are as follows: Shifted Exponential ($a = 0.0000001$, $\mu = 15$) and Weibull ($a = 0.2$, $\mu = 2$, $\alpha = 0.5$) for both the datasets. Note that, for model parallel MRPT, $l_i = |S|$. We use a similar strategy to simulate straggling in data parallel MRPT with l_i set to the average of $|S|$ for the set of test queries.

C. Results

Our experimental results are provided in Tables II and also illustrated in Fig. 5. For completion, we also include our obtained recall values here:

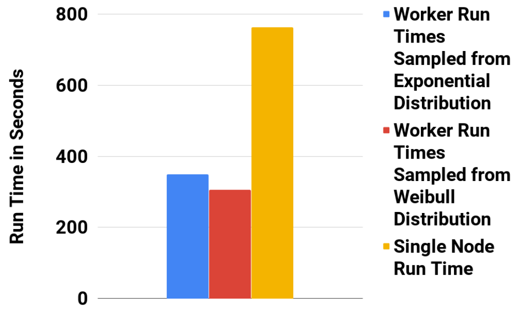
STL-10: Data Parallel (0.9632), Model Parallel (0.9648).

GIST: Data Parallel (0.9430), Model Parallel (0.9350).

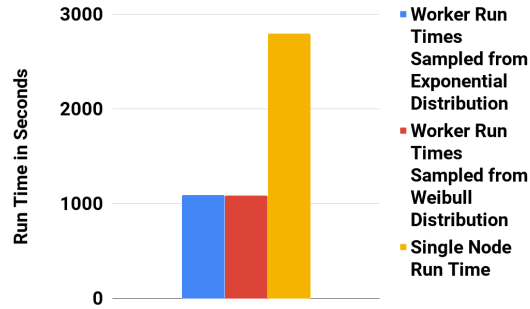
In all our experiments, both data and model parallel MRPT outperform the single node implementation. As shown in Fig. 5a and Fig. 5b, data parallel MRPT is significantly faster than single node MRPT. This is due to the smaller size of the MRPT index and absence of disk penalties at workers. It can be seen that the data parallel strategy has better performance

TABLE II: Runtime Statistics in seconds for MRPT on STL-10 and GIST datasets over 50 Runs

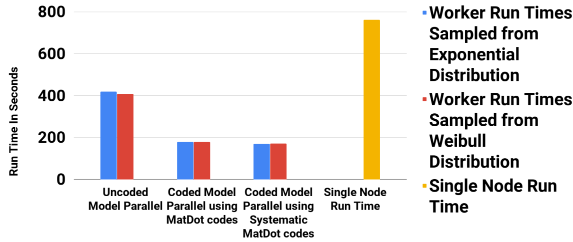
Experiment Type	*Mean (STL-10)*	*Std. Dev.(STL-10)*	*Mean (GIST)*	*Std. Dev.(GIST)*
Single Node	762.272	69.147	2795.791	173.395
Data Parallel, Shifted-Exponential Runtime	349.309	14.672	1091.010	44.261
Data Parallel, Weibull Runtime	305.046	14.44	1082.638	39.081
Uncoded Model Parallel, Shifted-Exponential Runtime	419.114	17.217	1260.504	49.566
Uncoded Model Parallel, Weibull Runtime	408.814	13.625	1261.980	62.259
Coded Model Parallel (MatDot), Shifted-Exponential Runtime	179.082	7.956	608.496	38.406
Coded Model Parallel (MatDot), Weibull Runtime	179.075	8.312	618.515	39.270
Coded Model Parallel (Systematic MatDot), Shifted-Exponential Runtime	168.869	6.865	604.546	22.951
Coded Model Parallel (Systematic MatDot), Weibull Runtime	171.428	8.526	622.716	22.495



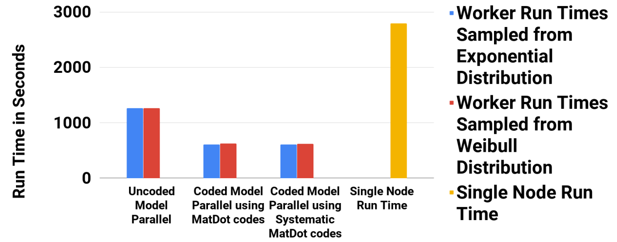
(a) Execution Times: Single Node & Data Parallel (STL-10).



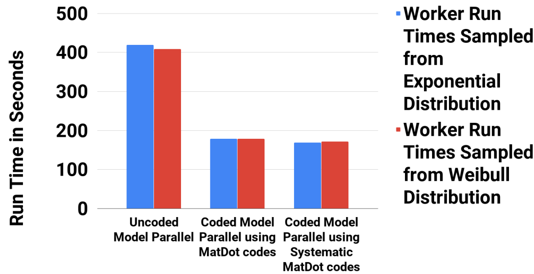
(b) Execution Times: Single Node & Data Parallel (GIST).



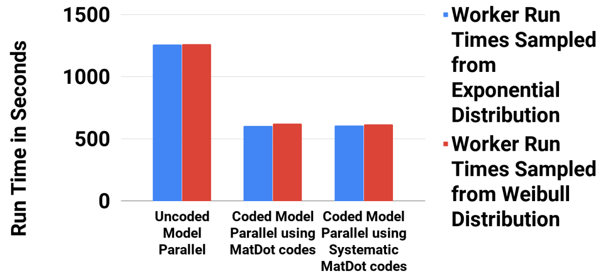
(c) Execution Times: Single Node & Model Parallel (STL-10).



(d) Execution Times: Single Node & Model Parallel (GIST).



(e) Effect of Coded Matrix Multiplication for STL-10.



(f) Effect of Coded Matrix Multiplication for GIST.

Fig. 5: Experimental Results of MRPT in different configurations.

when compared to uncoded model parallel strategy because of its embarrassingly parallel design. Owing to this design, the data parallel strategy could scale linearly with the number of nodes. However, these scaling benefits in query execution come at a cost, as the random projection trees computed at each node do not contain all the data points and hence the candidate set generated by each node could contain lesser true positives. To offset this condition, we might have to lower the voting threshold in-order to generate a better candidate, while causing more communication overheads and hence lower query execution time. The note mentioned in Section III explains this case in more detail. For our experiments, we do not lower the voting threshold as the loss in recall for STL-10 and GIST is not significant.

The model parallel architecture results in a high communication cost as the candidate set has to be transmitted to each worker node. However, we find that if the MRPT parameters for a dataset are sufficiently tuned, Algorithm 1 will generate

a smaller candidate set over which exact distance calculations must be performed. For our experiments, the algorithm was able to reduce the search space for STL-10 from 100000 datapoints to 3025 and for GIST from 1000000 to 14934 datapoints on average per query. Additionally, the model parallel strategy does not suffer from the accuracy related issues as compared to the data parallel architecture as the candidate set is generated using the entire dataset and not parts of it separately.

Both the strategies outperform baseline single node MRPT *despite running on inferior hardware*. The parallel strategies are not limited by memory constraints as in the case of single node MRPT. These strategies are therefore very useful when large datasets do not fit in memory. The coded model parallel strategy also makes the algorithm tolerant to slow nodes and failures in a distributed setting.

Fig. 5c and Fig. 5d show that model parallel MRPT outperforms single node MRPT. They also outperform data parallel MRPT under simulated straggling. This is of significance to real world systems where straggling may manifest as unreliable nodes or network delays. Fig. 5e and Fig. 5f show the benefits of coded matrix multiplication as opposed to the uncoded model parallel architecture under simulated straggling. Both MatDot codes and systematic MatDot codes are consistently faster than the uncoded approach. Fig. 5e also shows that systematic MatDot codes is able to outperform MatDot codes owing to its lower recovery threshold.

VI. CONCLUSIONS

We proposed two approaches to parallelize the MRPT algorithm in a distributed setting. We also applied the MatDot code based distributed matrix multiplication strategy to reduce the recovery threshold in a system that is prone to stragglers. We showed that our parallelization strategies can achieve faster queries than the single node MRPT algorithm under limited memory. Our results demonstrate the benefits of applying MatDot code and systematic MatDot code to the model parallel architecture in a system with simulated stragglers. In our experiments we observed large floating point errors when inverting high degree Vandermonde matrices for polynomial interpolation. As future work, we will experiment with strategies to reduce the condition number of a Vandermonde matrix [30] so that we can employ polynomials of higher degrees, and thus apply the MatDot code and systematic MatDot code with a larger number of worker nodes. Another possibility is to perform the computations in exact rational arithmetic. This would eliminate rounding errors, but the effect on runtime needs to be analyzed.

REFERENCES

[1] V. Hyvönen, T. Pitkänen, S. Tasoulis, E. Jääsaari, R. Tuomainen, L. Wang, J. Corander, and T. Roos, "Fast nearest neighbor search through sparse random projections and voting," in *IEEE Big Data*, 2016, pp. 881–888.

[2] Z. Xiong, W. Luo, L. Chen, and L. M. Ni, "Data vitalization: a new paradigm for large-scale dataset analysis," in *IEEE ICPADS*, 2010, pp. 251–258.

[3] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *ACM symposium on Theory of computing*, 1998, pp. 604–613.

[4] W. Cen and K. Miao, "An improved algorithm for locality-sensitive hashing," in *IEEE ICCSE*, 2015, pp. 61–64.

[5] B. Zhang and S. N. Srihari, "Fast k-nearest neighbor classification using cluster-based trees," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 26, no. 4, pp. 525–528, 2004.

[6] S. Liang, C. Wang, Y. Liu, and L. Jian, "CUKNN: A parallel implementation of k-nearest neighbor on CUDA-enabled GPU," in *IEEE Youth Conference on Information, Computing and Telecommunication*, 2009, pp. 415–418.

[7] L. Huang, Z. Liu, Z. Yan, P. Liu, and Q. Cai, "An implementation of high performance parallel KNN algorithm based on GPU," in *IEEE ICNDC*, 2012, pp. 30–30.

[8] M. M. A. Patwary *et al.*, "PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures," in *IEEE Parallel and Distributed Processing Symposium*, 2016, pp. 494–503.

[9] Z. Yu, Y. Liu, X. Yu, and K. Q. Pu, "Scalable distributed processing of K nearest neighbor queries over moving objects," *IEEE Trans. on Knowledge and Data Engineering*, vol. 27, no. 5, pp. 1383–1396, 2015.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[11] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, no. 2, pp. 22–28, 2003.

[12] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans. on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[13] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[14] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.

[15] D. Wang, G. Joshi, and G. Wornell, "Using Straggler Replication to Reduce Latency in Large-scale Parallel Computing," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 3, 2015, pp. 7–11.

[16] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *NIPS*, 2016, pp. 2092–2100.

[17] V. Cadambe and P. Grover, "Codes for Distributed Computing: A Tutorial," *IEEE Inf. Theory Newsletter*, vol. 67, no. 4, pp. 3–15, 2017.

[18] Y. Yang, P. Grover, and S. Kar, "Computing Linear Transformations With Unreliable Components," *IEEE Trans. on Information Theory*, vol. 63, no. 6, 2017.

[19] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *ICML*, 2017, pp. 3368–3376.

[20] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler Mitigation in Distributed Optimization through Data Encoding," in *NIPS*, 2017, pp. 5440–5448.

[21] Y. Yang, P. Grover, and S. Kar, "Coded Distributed Computing for Inverse Problems," in *NIPS*, 2017, pp. 709–719.

[22] H. Jeong, T. M. Low, and P. Grover, "Coded FFT and Its Communication Overhead," *Comm., Control, and Computing (Allerton)*, 2018.

[23] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," in *Comm., Control, and Computing (Allerton)*, 2017, pp. 1264–1270.

[24] S. Dutta, Z. Bai, H. Jeong, T. M. Low, and P. Grover, "A Unified Coded Deep Neural Network Training Strategy based on Generalized PolyDot codes," in *ISIT*, 2018, pp. 1585–1589.

[25] "Amazon AWS," http://aws.amazon.com/.

[26] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *AISTATS*, 2011, pp. 215–223.

[27] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[28] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," in *ISIT*, 2017, pp. 2403–2407.

[29] A. Reisizadeh, S. Prakash, R. Pedarsani, and S. Avestimehr, "Coded computation over heterogeneous clusters," in *ISIT*, 2017, pp. 2408–2412.

[30] E. E. Tyrtyshnikov, "How bad are hankel matrices?" *Numerische Mathematik*, vol. 67, no. 2, pp. 261–269, 1994.