Credence: Algorithm Based Fault Tolerance at Datacenter Scale

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Networking

Utsav Sheth

B.E., Computer Science and Engineering, R.V College of Engineering
M.S., Information Networking, Information Networking Institute, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

December, 2018

Acknowledgements

First and foremost, I want to thank Prof. Pulkit Grover for being an exceptionally supportive and enthusiastic advisor over the last few months. I have been working with Pulkit for a significant portion of my studies, and owe a great deal of my development to him. I would also like to thank Prof. Gauri Joshi for always having the time to have a conversation whenever I stopped by her office unannounced. I have also been fortunate to work with Haewon Jeong, Sanghamitra Datta. Yaoqing Yang, Malhar Chaudhari, and Ankur Mallick. They were always available at even the most inconvenient of hours to answer the most trivial questions. I would also like to thank the Information Networking Institute for their guidance and help throughout this process.

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

Abstract

As an increasing number of modern big data systems utilize horizontal scaling, the general trend in the distributed systems world has been to use general purpose com- modity hardware to reduce capital expenditure. System failures resulting from the use of inferior hardware have therefore become common at scale. Further, congested datacenter networks can result in high communication latencies and packet drops at network switches. Coded computing is a novel computing technique based on error correcting codes that aims to achieve algorithm based fault tolerance in a distributed system that is composed of unreliable compute nodes and networks. In this thesis, we explore the application of coded computing techniques to the problem of distributed matrix multiplication. Matrix multiplication is foundational to a number of applications today ranging from machine learning to scientific computing. We discuss some applications of coded matrix multiplication and then discuss the design and implementation of a Mesos framework that utilizes coded computing for distributed matrix multiplication and the methodology used to evaluate it. Finally, we discuss a novel scheduling strategy to minimize the latency of coded matrix multiplication jobs.

Table of Contents

List of Tables

List of Figures

1

Introduction

We are entering an era of ever-increasing data and computational requirements, paving the way towards increased distributed and parallel processing. For instance, nowadays, neural networks with millions of parameters [28, 43, 17] are becoming increasingly ubiquitous. To meet such requirements, supercomputing as well as cloud infrastructure are being leveraged constantly to support massive parallelism. However, one of the main challenges of large scale computing is to ensure reliability at scale [15, 12, 41, 2, 26]. The unreliability in massively distributed computing can be attributed to various factors, such as straggling, failures and soft errors:

- Straggling: The latency of the distributed computing system is often bottlenecked by a few slow workers known as *"stragglers"* [3, 33]. For instance, it was reported that stragglers are the biggest limiting factor when implementing distributed coordinate descent in the internal Google cloud [38].
- Faults, failures, soft-errors: For circuits and systems, faults and soft-errors [55, 36] are believed to be one of the main causes of the saturation of Moore's law on computation capacity. In the context of cloud computing, thousands of machine

and hard disk failures have been detected in newly configured clusters [2].

The primary objective of the system proposed in this thesis is to provide reliability and predictable performance in the face of stragglers. It is able to meet this objective by applying strategies derived from the field of *"Coded Computing"*. *Coded Computing* is an emerging direction of research that was first proposed to mitigate stragglers and reduce tail latency [32] in large scale distributed systems. Coded computing techniques build upon, and often improve on, the *Algorithm Based Fault Tolerance (ABFT)* [20, 18] paradigm. While ABFT techniques were originally employed to improve the circuit-level reliability of matrix multiplication operations, coded computing techniques employ error-correcting and erasure codes that are originally used for communication and storage systems respectively to make computation tolerant to faults and stragglers. There have been various research efforts in the systems community to reduce the tail latency by introducing redundancy [47, 11, 49, 42], but most strategies are based on naive replication. To protect against one node failure in a task the runs on 10 nodes, replication requires at least adding one replica for each node, resulting in the computation being carried out on 20 nodes. This implies that the overhead and resource requirement for the task is doubled under replication. A simple coded computing strategy using a (11,10) MDS code [30] would reduce the number of nodes required by 45% while offering the same amount of redundancy as replication. Other systems level solutions to straggler mitigation are based on the stale-synchronous parallel machine learning strategy [16].

In recent years, there have been significant theoretical advances in the field of coded computing. For example, a provably optimal redundant computing strategy for matrix-matrix multiplication was proposed in [9]. Coded computing techniques are applied to machine learning problems in [25, 7, 44]. However, the experimental testing of the existing coded computing techniques are often ad-hoc and applicationspecific and the effectiveness of coded computing in real-world general purpose distributed systems has not yet been evaluated. Different proposed techniques target different types of computation frameworks such as the standard master-worker framework [30], shared-memory systems [31], or fully-distributed systems [34, 22, 8, 7, 50]. Some existing frameworks for coded computing experimnets include HT-condor [6] or MPI-based platforms on Amazon EC2 [30].

1.1 Contributions

In this work, we propose a novel, cloud-based general purpose platform called *Credence* that enables researchers across different domain areas, from theorists to practitioners, to easily design and run coded computing experiments. The main benefits of the Credence platform are the following:

- Credence provides a cloud-based platform for reliable computing, that can scale across multiple nodes. Currently it provides deployment support for AWS but can be easily extended to other cloud providers, using their respective deployment APIs. Credence makes no assumptions on the underlying hardware or network topology unlike MPI. It is therefore portable and simple to use in different cloud environments.

- Credence as a platform is extensible and can incorporate newer coding strategies and techniques. Its architecture exposes standard programming interfaces to its users, which internally could work with any number of coding schemes and manipulation strategies.

- Credence uses a driver program written by the user. This program is a series of manipulations that the users want to perform on their data. It therefore abstracts away the complexities associated with distributed encoding and decoding and cluster management from the users, leaving them to focus solely on the implementation of their algorithms.

This thesis is laid out as follows. First, we review recent coded computing techniques: polynomial codes [51], MatDot codes, and generalized PolyDot codes [7]. We then motivate coded computing using real world applications. Next, we describe the design and implementation of Credence and then evaluate the performance of coded matrix multiplication on the system. Finally, we introduce the design of a scheduling algorithm for coded matrix multiplication jobs.

1.2 A Primer on Coded Matrix Multiplication

1.2.1 Distributed Matrix Multiplication Without Codes

Before we introduce the coded matrix multiplication strategy, we first describe a commonly used distributed matrix multiplication technique that does not utilize coded computing. Consider the problem of computing the product $\mathbf{C} = \mathbf{A}\mathbf{B}$ where $\mathbf{A}$ and $\mathbf{B}$ are N-by-N matrices. Given a grid of n-by-n worker nodes, the matrices $\mathbf{A}$ and $\mathbf{B}$ can be partitioned into N/n-by-N/n chunks and distributed across the grid of workers. Each worker at the grid position (r, c) can compute an outer product as follows:

$$C_{r,c} = \sum_{p=0}^{c-1} A_{r,p} B_{p,c} \tag{1.1}$$

Note that if any of the workers in the grid fails, the result matrix $\mathbf{C}$ cannot be retrieved. This is the fundamental idea in SUMMA [45]. This strategy is also known as *blocked matrix multiplication.* We will refer to this technique for the remainder of this thesis as *uncoded matrix multiplication.*

1.2.2 Coded Matrix Multiplication

We now describe the coded matrix multiplication strategies proposed in [10] to support coded matrix multiplication in *Credence*. We assume that we have a cluster consisting of a *master* node and P *worker nodes* and that we want to compute $\mathbf{C} = \mathbf{AB}$ where $\mathbf{A}$ and $\mathbf{B}$ are N-by-N matrices. A worker node has limited memory and computing power, so each node can receive the $1/m$-th fraction of matrices $\mathbf{A}$ and $\mathbf{B}$. Once a worker node completes its assigned computation, it reports the result back to *a fusion node.* Under this system model, *recovery threshold* is defined as the minimum number of workers required to recover the final computation output. The system model is discussed further in Section 3.2.

MatDot Codes

In this coded matrix multiplication strategy, the matrix $\mathbf{A}$ is partitioned vertically and $\mathbf{B}$ is partitioned horizontally as shown below,

$$\mathbf{A} = [\mathbf{A}_1\ \mathbf{A}_2\ \ldots\ \mathbf{A}_m], \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \vdots \\ \mathbf{B}_m \end{bmatrix}, \tag{1.2}$$

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

The matrices $\mathbf{A}$ and $\mathbf{B}$ are then encoded using the following polynomials:

$$p^M_{\mathbf{A}}(x) = \sum_{i=1}^{m} \mathbf{A}_i x^{i-1}, \quad p^M_{\mathbf{B}}(x) = \sum_{j=1}^{m} \mathbf{B}_i x^{m-j}. \tag{1.3}$$

A master node distributes encoded matrices, $p^M_{\mathbf{A}}(\alpha_c)$ and $p^M_{\mathbf{B}}(\alpha_c)$ to the c-th worker node $(c = 1, \cdots, P)$. Then the c-th worker node computes the following product at

$x = \alpha_c$:

$$p_{\mathbf{C}}^{M}(x) = P_A^M(x)P_B^M(x) = \sum_{i=1}^{m}\sum_{j=1}^{m} \mathbf{A}_i\mathbf{B}_j x^{m-1+(i-j)}, \tag{1.4}$$

and returns the result to a fusion node. Note that the coefficient of x^{m-1} in equation 1.4 is $\mathbf{C} = \sum_{i=1}^{m} A_i B_i$. Since the degree of the polynomial $p_{\mathbf{C}}(x)$ is $2m-2$, once the fusion node receives the evaluation of $p_{\mathbf{C}}(x)$ at any $2m-1$ distinct points, it can recover the coefficients of $p_{\mathbf{C}}^{M}(x)$. The recovery threshold is therefore $K = 2m - 1$. This was proven in [10] to be the optimal recovery threshold for the given storage constraint that a worker node can store $1/m$-th fraction of each input matrix [52].

Systematic MatDot Codes

Although Credence does not support *Systematic MatDot Codes*, we have found this coding strategy to be useful in practice. 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 $p_{\mathbf{A}}^{S}(x) = \sum_{i=1}^{m} \mathbf{A}_i L_i(x)$ and $p_{\mathbf{B}}^{S}(x) = \sum_{i=1}^{m} \mathbf{B}_i L_i(x)$ where $L_i(x)$ is defined as follows for $i \in \{1, \ldots, m\}$:

$$L_i(x) = \prod_{j\in\{1,\ldots,m\}\setminus\{i\}} \frac{x - x_j}{x_i - x_j}. \tag{1.5}$$

Using these polynomials, the worst-case recovery threshold remains the same as non-systematic MatDot codes [10].

Generalized PolyDot Codes

Polynomial codes [51] have a recovery threshold of m^2 and a communication cost of $\mathcal{O}(N^2/m^2)$, wheres MatDot codes have a lower recovery threshold of $2m-1$, but a higher communication cost of $\mathcal{O}(N^2)$ per node. *PolyDot* codes proposed in [10] provide an intermediate recovery threshold and communication cost between Polynomial codes and MatDot codes. In this coded matrix multiplication strategy, the matrices $\mathbf{A}$ and $\mathbf{B}$ are partitioned both horizontally and vertically as shown below,

$$A = \begin{bmatrix} A_{0,0} & \dots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \dots & A_{m-1,n-1} \end{bmatrix}, \quad B = \begin{bmatrix} B_{0,0} & \dots & B_{0,d-1} \\ \vdots & \ddots & \vdots \\ B_{n-1,0} & \dots & B_{n-1,d-1} \end{bmatrix} \tag{1.6}$$

The matrices $\mathbf{A}$ and $\mathbf{B}$ are then encoded using the following polynomials:

$$p_{\mathbf{A}}(u,v) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} \mathbf{A}_{i,j} u^i v^j, \quad p_{\mathbf{B}}(v,w) = \sum_{j=0}^{n-1}\sum_{k=0}^{d-1} \mathbf{B}_{i,j} v^{n-1-j} w^k \tag{1.7}$$

As before, a master node distributes encoded matrices, $p_{\mathbf{A}}(u_c, v_c)$ and $p_{\mathbf{B}}(v_c, w_c)$ to the c-th worker node ($c = 1, \cdots, P$). Then the c-th worker node then computes the following product at $x = \alpha_c$:

$$p_{\mathbf{C}}(u,v,w) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\sum_{j'=0}^{n-1}\sum_{k=0}^{d-1} \boldsymbol{A}_{i,j}\boldsymbol{B}_{j',k} u^i v^{n-1+j-j'} w^k \tag{1.8}$$

Now, fixing $j' = j$, we observe that the coefficient of $u^i v^{n-1} w^k$ for $i = 0, 1, \dots, m-1$ and $k = 0, 1, \dots, d-1$ turns out to be $\sum_{j=0}^{n-1} \boldsymbol{A}_{i,j}\boldsymbol{B}_{j,k} = \boldsymbol{C}_{i,k}$. These md coefficients constitute the $m \times d$ sub-matrices (or blocks) of $\boldsymbol{C} = \boldsymbol{AB}$. Therefore, $\boldsymbol{C}$ can be recovered at the decoder if all these md coefficients of the polynomial $p_{\mathbf{C}}(u,v,w)$ can be interpolated from its evaluations at different nodes.

For garbage alignment, Dutta et al. [8] propose the substitution ($u = v^n, w = v^{mn}$) to convert $p_{\mathbf{C}}(u,v,w)$ into a polynomial of a single variable v. The polynomial therefore reduces to:

$$\widetilde{\boldsymbol{S}}(v) = \widetilde{\boldsymbol{S}}(u,v,w)|_{u=v^n, w=v^{mn}} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\sum_{j'=0}^{n-1}\sum_{k=0}^{d-1} \boldsymbol{W}_{i,j}\boldsymbol{X}_{j',k} v^{ni+mnk+n-1+j-j'} \tag{1.9}$$

which is a polynomial in a single variable v. Its degree is given by $n(m-1) + mn(d-1) + n - 1 + n - 1 = mnd + n - 2$. Thus, the fusion node needs to wait for $mnd + n - 1$ nodes, each producing a unique evaluation, to be able to interpolate all its $mnd + n - 1$ coefficients. Note that by substituting $m = 1$, $n = n$, and $d = 1$, Generalized PolyDot specializes to MatDot.

2

Coded Computing in the Real World

2.1 Introduction

This chapter motivates coded computing by introducing some examples of coded computing applied to real world problems in machine learning. Coded implementations of k-nearest neighbors estimation, linear regression, and deep neural network training are analyzed in this chapter.

2.2 Fast k-NN Estimation Using MatDot Codes

In this section, 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 the MRPT algorithm for k-NN estimation [21] 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 [21]) including genomics, personalized search, network security, and web based recommendation systems. We formulate a coded computing problem for MRPT, and then apply the coded matrix multiplication strategy MatDot codes to further reduce the query time for the distributed architecture in a system that is prone to straggling.

The MRPT Algorithm

This section briefly describes the two stages of the MRPT algorithm: (i) off-line index construction stage and (ii) on-line 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 $dist(\mathbf{x}, \mathbf{q}) \leqslant dist(\mathbf{y}, \mathbf{q})$ 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:

$$dist(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\| = \sqrt{\|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\mathbf{u} \cdot \mathbf{v}} \tag{2.1}$$

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

Index Construction in MRPT

In the MRPT index construction phase, 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 [21], 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 $1 \leqslant t \leqslant T$, let $f_t(\cdot)$ be defined as:

$$f_t(\mathbf{x} : \mathbf{q}) = \sum_{i=1}^{2^\ell} 1(\mathbf{x} \in L_i, \mathbf{q} \in L_i) \tag{2.2}$$

where $1(\mathbf{x} \in L_i, \mathbf{q} \in L_i)$ 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:

$$F(\mathbf{x}; \mathbf{q}) = \sum_{t=1}^{T} f_t(\mathbf{x}, \mathbf{q}). \tag{2.3}$$

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

$$S = \{j \subset \{1, 2, \ldots, N\} : \mathbf{x}_j \in \mathcal{X} \text{ and } F(\mathbf{x}_j; \mathbf{q}) \geqslant \nu\} \tag{2.4}$$

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, `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, `TREE_QUERY`$(\mathbf{q}, t)$ gives $\{j \subset \{1, \ldots, N\} : 1(\mathbf{x}_j \in L_i, \mathbf{q} \in L_i) = 1$ for some $L_i\}$. The exact distance calculation is performed as shown in Equation (2.1).

Algorithm 1 The MRPT Query Phase

```
procedure APPROXIMATE_KNN(q, k, $\mathcal{T}$, $\nu$)
    $S \leftarrow \varnothing$
    Let votes = [0, ..., 0] be a new $n$-dimensional array
    for $t$ in $\mathcal{T}$ do
        for point in TREE_QUERY(q, t) do
            votes[point] ← votes[point] + 1
            if votes[point] = $\nu$ then
                $S \leftarrow S \cup \{point\}$
    return EXACT_KNN(q, k, S)
```

Problem Formulation for Distributed MRPT

Consider the d-dimensional dataset $\mathcal{X}$ as before. In the distributed 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 2.2. 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 (2.1), 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 $j \in S$.

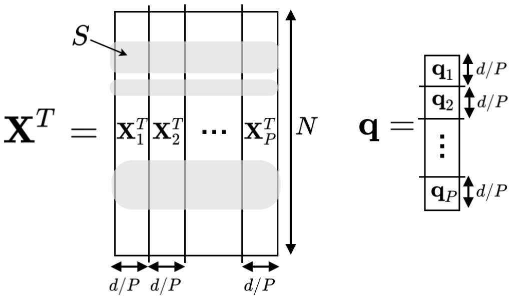

Figure 2.1: Partitioning the data matrix $\mathbf{X}$ and the query vector $\mathbf{q}$ in Distributed MRPT.

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 (2.1) 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 strategies that compute vector $\mathbf{w}$ in a distributed setting.

Uncoded Distributed Matrix-Vector Multiplication

We split $\mathbf{X}^T$ into P equal partitions as follows (see Figure 2.1):

$$\mathbf{X}^T = \begin{bmatrix}\mathbf{X}_1^T & \mathbf{X}_2^T & \dots & \mathbf{X}_P^T\end{bmatrix}. \tag{2.5}$$

Now consider a cluster consisting of one *master node* and P *worker nodes* as shown in Fig. Figure 2.2. 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).

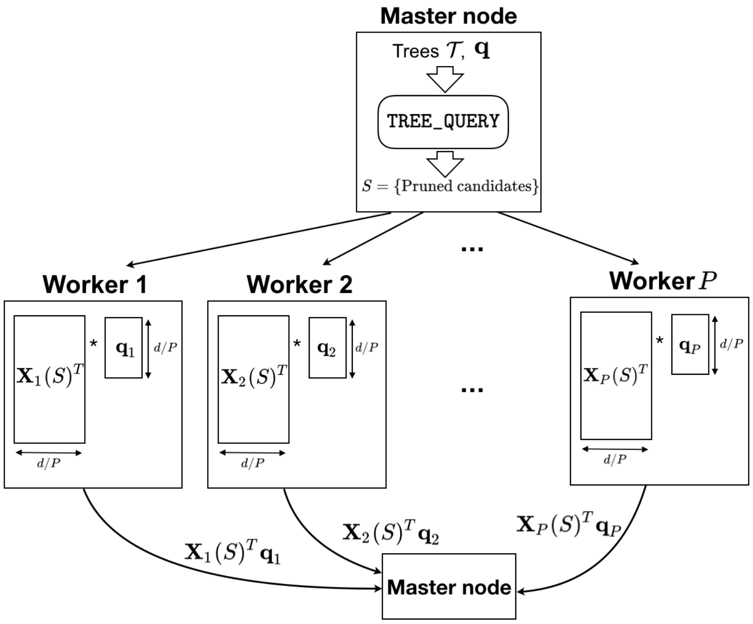

Figure 2.2: Architecture with Uncoded Distributed Matrix Multiplication.

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

$$\mathbf{X}(S)^T = \begin{bmatrix}\mathbf{X}_1(S)^T & \mathbf{X}_2(S)^T & \dots & \mathbf{X}_P(S)^T\end{bmatrix} \tag{2.6}$$

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 Figure 2.1). The product $\mathbf{w}$ can then be expressed as:

$$\mathbf{w} = \sum_{i=1}^{P} \mathbf{X}_i(S)^T\mathbf{q}_i \tag{2.7}$$

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 Equation (2.7), and determine the k nearest neighbors using the exact distances.

Coded Distributed Matrix-Vector Multiplication using MatDot Codes

In order to successfully compute the vector $\mathbf{w}$ using the strategy described in Section 2.2, 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 [10].

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

$$\mathbf{X}^T = \begin{bmatrix} \mathbf{X}_1^T & \mathbf{X}_2^T & \dots & \mathbf{X}_m^T. \end{bmatrix} \tag{2.8}$$

We then use the following encoding polynomial:

$$P_{\mathbf{X}^T}(\beta) = \sum_{j=1}^{m} \mathbf{X}_j^T \beta^{j-1}. \tag{2.9}$$

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

$$P_{\mathbf{X}^T(S)}(\beta) = \sum_{j=1}^{m} \mathbf{X}_j(S)^T \beta^{j-1}. \tag{2.10}$$

We will be referring to this observation later.

Now, given a cluster with a master node and P worker nodes, as shown in Figure 2.3, 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 (2.9). This encoding step can be performed off-line as $\mathbf{X}^T$ is known in advance.

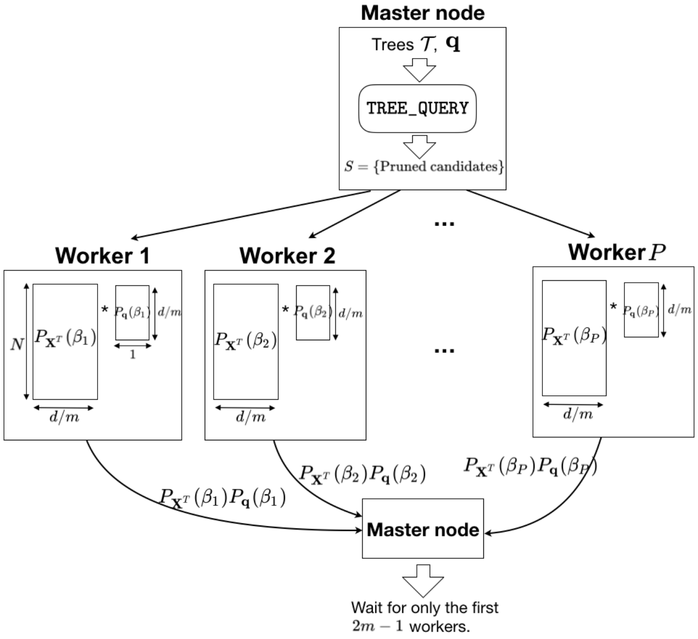
Master node
Trees $\mathcal{T}$, $\mathbf{q}$
TREE_QUERY
$S = \{\text{Pruned candidates}\}$
...
Worker 1
Worker 2
Worker P
N
$P_{\mathbf{X}^T}(\beta_1)$
$P_{\mathbf{q}}(\beta_1)$
d/m
1
$P_{\mathbf{X}^T}(\beta_2)$
$P_{\mathbf{q}}(\beta_2)$
$P_{\mathbf{X}^T}(\beta_P)$
$P_{\mathbf{q}}(\beta_P)$
$P_{\mathbf{X}^T}(\beta_1)P_{\mathbf{q}}(\beta_1)$
$P_{\mathbf{X}^T}(\beta_2)P_{\mathbf{q}}(\beta_2)$
$P_{\mathbf{X}^T}(\beta_P)P_{\mathbf{q}}(\beta_P)$
Master node
Wait for only the first $2m-1$ workers.

Figure 2.3: Distributed Architecture with Coded Matrix Multiplication Using Mat-Dot Codes.

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: $\{\mathbf{q}_j : j \in \{1, 2, \ldots, m\}\}$. We then use the following encoding polynomial:

$$P_{\mathbf{q}}(\beta) = \sum_{j=1}^{m} \mathbf{q}_j \beta^{m-j} \tag{2.11}$$

As in Section 2.2, the master node first determines the candidate set S. It then transmits S and the encoded query $P_{\mathbf{q}}(\beta_i)$ obtained from Equation 2.11 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 Equation (2.10)) 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

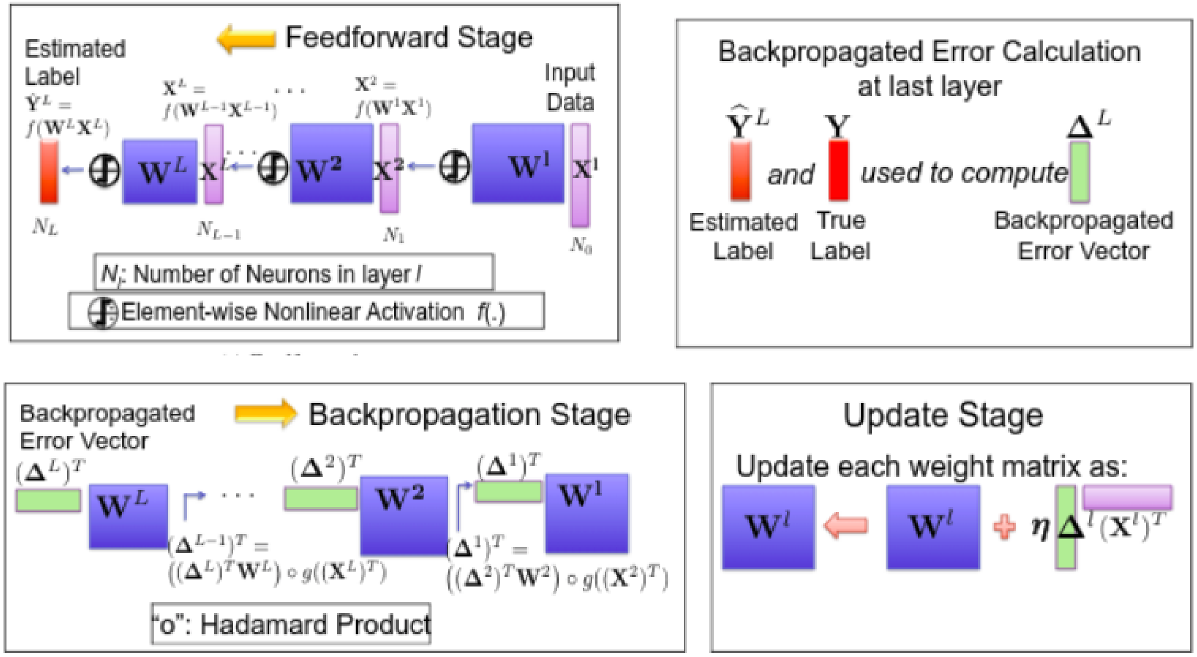
Feedforward Stage
Estimated Label
$\hat{\mathbf{Y}}^L = f(\mathbf{W}^L\mathbf{X}^L)$
$\mathbf{X}^L = f(\mathbf{W}^{L-1}\mathbf{X}^{L-1})$
$\mathbf{X}^2 = f(\mathbf{W}^1\mathbf{X}^1)$
Input Data
$\mathbf{W}^L$
$\mathbf{X}^L$
$\mathbf{W}^2$
$\mathbf{X}^2$
$\mathbf{W}^1$
$\mathbf{X}^1$
N_L
N_{L-1}
N_1
N_0
N_l: Number of Neurons in layer l
Element-wise Nonlinear Activation $f(.)$
Backpropagated Error Calculation at last layer
$\hat{\mathbf{Y}}^L$ and $\mathbf{Y}$ used to compute $\mathbf{\Delta}^L$
Estimated Label
True Label
Backpropagated Error Vector
Backpropagated Error Vector
Backpropagation Stage
$(\mathbf{\Delta}^L)^T$
$\mathbf{W}^L$
$(\mathbf{\Delta}^{L-1})^T = ((\mathbf{\Delta}^L)^T\mathbf{W}^L) \circ g((\mathbf{X}^L)^T)$
$(\mathbf{\Delta}^2)^T$
$\mathbf{W}^2$
$(\mathbf{\Delta}^1)^T$
$\mathbf{W}^1$
$(\mathbf{\Delta}^1)^T = ((\mathbf{\Delta}^2)^T\mathbf{W}^2) \circ g((\mathbf{X}^2)^T)$
"o": Hadamard Product
Update Stage
Update each weight matrix as:
$\mathbf{W}^l$
$\mathbf{W}^l$
$+ \ \eta \mathbf{\Delta}^l (\mathbf{X}^l)^T$

Figure 2.4: Steps Involved in DNN Training

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 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 2.2 to obtain the set of the k nearest neighbors to $\mathbf{q}$.

2.3 Deep Neural Network Training Using Generalized PolyDot Codes

We assume that a DNN with L layers (excluding the input layer) is being trained using backpropagation with Stochastic Gradient Descent (SGD) with a mini-batch size of $B = 1$ [40]. The DNN thus consists of L weight matrices, one for each layer, that represent the connections between the l-th and $(l-1)$-th layer for $l = 1, 2, \ldots, L$.

At the l-th layer, N_l denotes the number of neurons. Thus, the weight matrix to be trained is of dimension $N_l \times N_{l-1}$. For simplicity of presentation, we assume that $N_l = N$ for all layers. In every iteration, the DNN (*i.e.* the L weight matrices) is trained based on a single data point and its true label through three stages, namely, feedforward, backpropagation and update, as shown in Figure 2.4. At the beginning of every iteration, the first layer accesses the data vector (input for layer 1) and starts the feedforward stage which propagates from layer $l = 1$ to $l = L$. For a layer l, let us denote the weight matrix, input for the layer and backpropagated error for iteration k by $\boldsymbol{W}^l(k)$, $\boldsymbol{x}^l(k)$ and $\boldsymbol{\delta}^l(k)$ respectively. The operations performed in layer l during feedforward stage can be summarized as:

- Compute matrix-vector product $\boldsymbol{s}^l(k) = \boldsymbol{W}^l(k)\boldsymbol{x}^l(k)$.
- Compute input for layer $(l+1)$ given by $\boldsymbol{x}^{(l+1)}(k) = f(\boldsymbol{s}^l(k))$ where $f(\cdot)$ is a nonlinear activation function applied elementwise.

At the last layer ($l = L$), the backpropagated error vector is generated by assessing the true label and the estimated label, $f(\boldsymbol{s}^L(k))$, which is output of last layer. Then, the backpropagated error propagates from layer L to 1, also updating the weight matrices at every layer alongside. The operations for the backpropagation stage can be summarized as:

- Compute matrix-vector product $[\boldsymbol{c}^l(k)]^T = [\boldsymbol{\delta}^l(k)]^T \boldsymbol{W}^l(k)$.
- Compute backpropagated error vector for layer $(l-1)$ given by $[\boldsymbol{\delta}^{(l-1)}(k)]^T = [\boldsymbol{c}^l(k)]^T \boldsymbol{D}^l(k)$ where $\boldsymbol{D}^l(k)$ is a diagonal matrix whose i-th diagonal element depends only on the i-th value of $\boldsymbol{x}^l(k)$. More specifically, $\boldsymbol{D}^l(k)$ is a diagonal matrix whose i-th diagonal element is a function $g(\cdot)$ of the i-th element of $\boldsymbol{x}^l(k)$, such that, $g(f(u)) = f'(u)$ for the chosen nonlinear activation function

$f(\cdot)$ in the feedforward stage. This is equivalent to computing the Hadamard product: $[\boldsymbol{\delta}^{(l-1)}(k)]^T = [\boldsymbol{c}^l(k)]^T \circ g([\boldsymbol{x}^l(k)]^T)$.

Finally, the step in the update stage is as follows:

- Update as: $\boldsymbol{W}^l(k+1) \leftarrow \boldsymbol{W}^l(k) + \eta \boldsymbol{\delta}^l(k)[\boldsymbol{x}^l(k)]^T$ where η is the learning rate.

In DNN training with PolyDot codes, the three stages are modified to support coded matrix multiplication. Prior to training, every node stores an $\frac{N}{m} \times \frac{N}{n}$ sub-matrix (or block) of $\boldsymbol{W}$ encoded using generalized PolyDot. Recall from Equation (1.7) that,

$$\widetilde{\boldsymbol{W}}(u,v) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \boldsymbol{W}_{i,j} u^i v^j. \tag{2.12}$$

Thus every node stores a sub-matrix $\widetilde{\boldsymbol{W}}_p = \widetilde{\boldsymbol{W}}(a_p, b_p)$ at the beginning of the training. Each input $\boldsymbol{x}$ to the DNN is encoded at each worker as follows:

$$\widetilde{\boldsymbol{x}}(v) = \sum_{j=0}^{n-1} \boldsymbol{x}_j v^{n-j-1}. \tag{2.13}$$

Thus every node stores a sub-matrix $\widetilde{\boldsymbol{x}}_p = \widetilde{\boldsymbol{x}}(b_p)$ at the beginning of the training. We will refer to $\widetilde{\boldsymbol{W}}_p$ and $\widetilde{\boldsymbol{x}}_p$ at the k-th iteration as $\widetilde{\boldsymbol{W}}(k)$ and $\widetilde{\boldsymbol{x}}(k)$ respectively. The steps performed during the feedforward stage can be summarized as:

- Compute matrix-vector product $\widetilde{\boldsymbol{s}}(k) = \widetilde{\boldsymbol{W}}(k).\widetilde{\boldsymbol{x}}(k)$.

- Wait for the recovery threshold for this operation to be met and decode $\widetilde{\boldsymbol{s}}(k)$ to get $s^l(k)$.

- Compute input for layer $(l+1)$ given by $\boldsymbol{x}^{(l+1)}(k) = f(\boldsymbol{s}^l(k))$ where $f(\cdot)$ is a nonlinear activation function applied elementwise.

Similar to the feedforward stage, every worker encodes the backpropagated error (transpose) $\boldsymbol{\delta}^T$ is available at every worker. Each worker then splits $\boldsymbol{\delta}^T$ into m equal parts and encodes them using the polynomial:

$$\widetilde{\boldsymbol{\delta}}^T(u) = \sum_{i=0}^{m-1} \boldsymbol{\delta}_i^T u^{m-i-1}. \tag{2.14}$$

In the backpropagation stage in coded DNN training, for $p = 0, 1, \ldots, P-1$, the p-th node does the following:

- Computes $\widetilde{\boldsymbol{\delta}}^T(u)$ at $u = a_p$, yielding $\widetilde{\boldsymbol{\delta}}_p^T := \widetilde{\boldsymbol{\delta}}^T(a_p)$. Next, it **performs** the computation $\widetilde{\boldsymbol{c}}_p^T := \widetilde{\boldsymbol{\delta}}_p^T \widetilde{\boldsymbol{W}}_p$. $\widetilde{\boldsymbol{\delta}}^T(u)$ at $u = a_p$, yielding $\widetilde{\boldsymbol{\delta}}_p^T := \widetilde{\boldsymbol{\delta}}^T(a_p)$.
- Next, it **performs** the computation $\widetilde{\boldsymbol{c}}_p^T := \widetilde{\boldsymbol{\delta}}_p^T \widetilde{\boldsymbol{W}}_p$. Consider the polynomial:

  $$\widetilde{\boldsymbol{c}}^T(u,v) = \widetilde{\boldsymbol{\delta}}^T(u)\widetilde{\boldsymbol{W}}(u,v) = \sum_{i'=0}^{m-1}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} \boldsymbol{\delta}_{i'}^T \boldsymbol{W}_{i,j} u^{m-1+i-i'} v^j.$$

  The products computed at each node result in the evaluations of this polynomial $\widetilde{\boldsymbol{c}}^T(u,v)$ at $(u,v) = (a_p, b_p)$. Similar to feedforward stage, each node then decodes the coefficients of $u^{m-1}v^j$ in the polynomial for $j = 0, 1, \ldots, n-1$, and thus reconstructs n sub-vectors forming $\boldsymbol{c}^T$.
- Computes backpropagated error vector for layer $(l-1)$ as in uncoded DNN training.

The key part is *updating* the coded $\widetilde{\boldsymbol{W}}_p$. Observe that since $\boldsymbol{x}$ and $\boldsymbol{\delta}$ are both available at each node, it can encode the vectors as $\sum_{i=0}^{m-1} \boldsymbol{\delta}_i u^i$ and $\sum_{j=0}^{n-1} \boldsymbol{x}_j v^j$ at

$u = a_p$ and $v = b_p$ respectively, and then update itself as follows:

$$\widetilde{\boldsymbol{W}}_p \leftarrow \widetilde{\boldsymbol{W}}_p + \eta(\sum_{i=0}^{m-1} \boldsymbol{\delta}_i a_p^i)(\sum_{j=0}^{n-1} \boldsymbol{x}_j b_p^j)^T$$

$$= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} \underbrace{(\boldsymbol{W}_{i,j} + \eta\boldsymbol{\delta}_i\boldsymbol{x}_j^T)}_{\text{Update of } \boldsymbol{W}_{i,j}} a_p^i b_p^j. \tag{2.15}$$

Thus, the update step preserves the coded nature of the weight matrix, with negligible additional overhead [8] and the coded weight matrices do not have to be decoded, modified, and re-encoded .

2.4 Multi-label Linear Regression Using Generalized PolyDot Codes

Here, we discuss a technique of coded linear regression and demonstrate how it can be implemented easily using generalized PolyDot codes. This technique can be obtained as a special case of the more detailed coded DNN training strategy. Similar to feedforward stage, each node then decodes the coefficients of $u^{m-1}v^j$ in the polynomial for $j = 0, 1, \dots, n-1$, and thus reconstructs n sub-vectors forming $\boldsymbol{c}^T$.

Problem Formulation:

We are given a dataset χ consisting of pairs of vectors $(\boldsymbol{x}, \boldsymbol{y})$ denoting the datapoint and its multi-dimensional label. We would like to train a matrix $\boldsymbol{W}$ over this dataset, such that:

$$\boldsymbol{W} = argmin_{\boldsymbol{W}} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\chi} \frac{1}{|\chi|}||\boldsymbol{W}\boldsymbol{x} - \boldsymbol{y}||^2. \tag{2.16}$$

The training is performed iteratively using Stochastic Gradient Descent (or its mini-batch variant). At each iteration, the algorithm picks up a pair $(\boldsymbol{x}, \boldsymbol{y})$ (or a mini-batch of such pairs), and updates the trained $\boldsymbol{W}$ as follows:

$$\begin{aligned}\boldsymbol{W} &= \boldsymbol{W} - \eta \nabla_{\boldsymbol{W}} ||\boldsymbol{W}\boldsymbol{x} - \boldsymbol{y}||_2^2 \\ &= \boldsymbol{W} - 2\eta(\boldsymbol{W}\boldsymbol{x} - \boldsymbol{y})\boldsymbol{x}^T \\ &= \boldsymbol{W} + 2\eta\boldsymbol{\delta}\boldsymbol{x}^T \text{ where } \boldsymbol{\delta} = \boldsymbol{y} - \boldsymbol{W}\boldsymbol{x}. \end{aligned} \tag{2.17}$$

We are provided with a system of P memory-constrained distributed worker nodes, such that, each node can only store a fixed fraction of $\boldsymbol{W}$. Additionally, the nodes are also prone to delays due to straggling. The goal is to be able to parallelize this iterative computation across P memory-constrained nodes along with redundancy, such that we do not have to wait for all the nodes to finish at each iteration.

We now demonstrate an example where each node only stores 1/4 fraction of matrix $\boldsymbol{W}$. The primary operations to be performed are the matrix-vector product $\boldsymbol{W}\boldsymbol{x}$, followed by the subtraction of vectors, *i.e.*, $\boldsymbol{\delta} = \boldsymbol{y} - \boldsymbol{W}\boldsymbol{x}$, and then the update of $\boldsymbol{W}$ using a rank-1 update rule ($\boldsymbol{W} + 2\eta\boldsymbol{\delta}\boldsymbol{x}^T$). At the beginning of the training, we choose an initial value for the parameter matrix $\boldsymbol{W}$. The matrix $\boldsymbol{W}$ is distributed across the workers which then encodes the matrix using the generalized PolyDot encoding strategy as shown in Equation (1.7). In particular, each node stores a unique evaluation of the polynomial (with $u = v^2$):

$$\widetilde{\boldsymbol{W}}(u, v) = \boldsymbol{W}_{0,0} + \boldsymbol{W}_{0,1}v + \boldsymbol{W}_{1,0}u + \boldsymbol{W}_{1,1}uv. \tag{2.18}$$

Because encoding of sub-matrices is expensive, we only encode these sub-matrices once initially at the beginning of training, and for subsequent iterations, we only perform coded updates instead of encoding the sub-matrices afresh. At every iteration, the vector $\boldsymbol{x}$ is also divided into 2 equal parts $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, and then encoded as:

$$\widetilde{\boldsymbol{x}}(v) = \boldsymbol{x}_0 v + \boldsymbol{x}_1. \tag{2.19}$$

Each node already stores a unique evaluation of $\widetilde{\boldsymbol{W}}(u, v)$ with $u = v^2$, and obtains an evaluation of $\widetilde{\boldsymbol{x}}(v)$ at the same value. It thus multiplies $\widetilde{\boldsymbol{W}}(u, v)$ with $\widetilde{\boldsymbol{x}}(v)$, resulting in the evaluation of the following polynomial:

$$\begin{aligned}\widetilde{\boldsymbol{W}}(u,v)\widetilde{\boldsymbol{x}}(v) &= (.) + (\boldsymbol{W}_{0,0}\boldsymbol{x}_0 + \boldsymbol{W}_{0,1}\boldsymbol{x}_1)v + (.)v^2 \\ &\quad + (\boldsymbol{W}_{1,0}\boldsymbol{x}_0 + \boldsymbol{W}_{1,1}\boldsymbol{x}_1)v^3 + (.)v^4. \end{aligned} \tag{2.20}$$

After this multiplication, each node stores its resulting shard from the output. As the polynomial is of degree 4, the cluster only has to wait for any 5 worker nodes to finish itself before it can interpolate all the coefficients of the polynomial, which also includes sub-vectors of the result $\boldsymbol{W}\boldsymbol{x}$.

Once the matrix-vector product $\boldsymbol{W}\boldsymbol{x}$ is computed, the cluster can compute the subtraction $\boldsymbol{\delta} = \boldsymbol{y} - \boldsymbol{W}\boldsymbol{x}$. After this step, each worker node locally divides $\boldsymbol{\delta}$ and $\boldsymbol{x}$ into 2 sub-vectors respectively, and performs local encoding of sub-vectors which is much cheaper than encoding matrices. The model is then updated as shown below:

$$\begin{aligned}\widetilde{\boldsymbol{W}}^{+}(u,v) &= \widetilde{\boldsymbol{W}}(u,v) + 2\eta(\underbrace{\boldsymbol{\delta}_0 + \boldsymbol{\delta}_1 u}_{\text{local encoding}})(\underbrace{\boldsymbol{x}_0 + \boldsymbol{x}_1 v}_{\text{local encoding}})^T \\ &= (\boldsymbol{W}_{0,0} + \boldsymbol{\delta}_0\boldsymbol{x}_0^T) + (\boldsymbol{W}_{0,1} + \boldsymbol{\delta}_0\boldsymbol{x}_1^T)v \\ &\quad + (\boldsymbol{W}_{1,0} + \boldsymbol{\delta}_1\boldsymbol{x}_0^T)u + (\boldsymbol{W}_{1,1} + \boldsymbol{\delta}_1\boldsymbol{x}_1^T)uv \\ &= \boldsymbol{W}_{0,0}^{+} + \boldsymbol{W}_{0,1}^{+}v + \boldsymbol{W}_{1,0}^{+}u + \boldsymbol{W}_{1,1}^{+}uv. \end{aligned}$$

This local encoding of sub-vectors, followed by local rank-1 update results in the update of the coded sub-matrix without having to encode the sub-matrix afresh. This process is then repeated iteratively for all $(\boldsymbol{x},\boldsymbol{y}) \in \chi$.

3

Credence: A Platform For Coded Computing

In this chapter, we describe the design and implementation of Credence on Mesos. We first describe how individual matrix multiplication jobs are run on Credence and then describe the Credence programming model.

3.1 An Introduction to Mesos

Mesos [19] is a platform for resource sharing between distributed systems, called *frameoworks*, at datacenter scale. It is different from other resource sharing systems such as YARN [46] in that it does not implement its own scheduler. The Mesos allocator is a pluggable module that uses the DRF resource allocation strategy [13] by default. Mesos makes offers to different frameworks leaves job scheduling decisions up to the framework itself. It is therefore employs a *push-based* allocation strategy. Distributed systems running on YARN, called *applications*, make resource requests to the YARN resource manager which internally implements an application scheduling policy. Employing a push-based policy in Mesos has the following advantages [19]: it allows frameworks to make decisions on scheduling and error and fault handling, and it makes the Mesos design simple and robust.

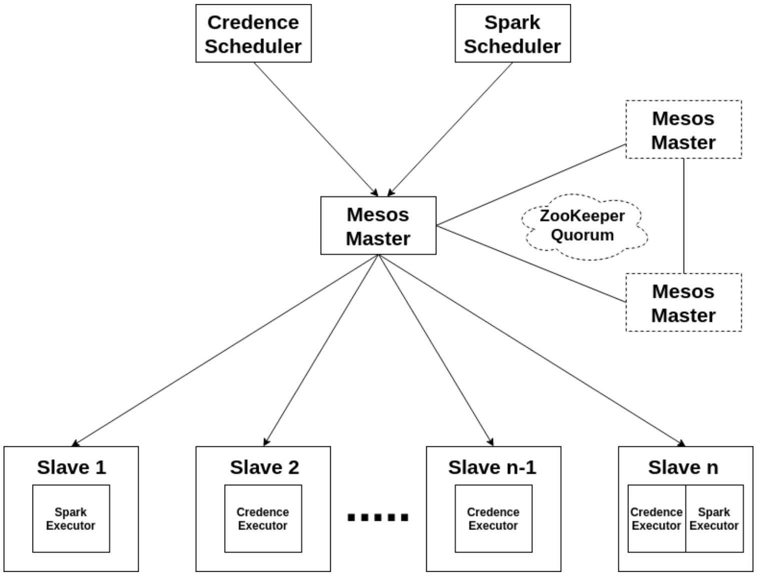

Figure 3.1: The Mesos Master-Slave Architecture.

A Mesos cluster consists of one *master* and one or more *slaves*, also called *agents*. Frameworks can utilize cluster resources by registering themselves with the Mesos master from within their schedulers. This may also involve an authentication step. Once registered, Mesos begins to make resource offers to the framework's scheduler at which point the framework can utilize the cluster's resources to run its jobs. Typically, jobs consist of tasks that are run in parallel on workers in a cluster. Classical examples of systems that follow this architecture are MapReduce [4] in Hadoop and Spark [54]. Mesos runs tasks inside independent units called *executors* on Mesos agents. Tasks may be executables, BASH or Python scripts, JAR files, and even Docker containers. Mesos implements complete isolation between executors running on the same agent by the use of Linux *cgroups*. Once executors terminate, resources are returned to Mesos and can now be offered to other frameworks. Frameworks running on Mesos are therefore not limited to a statically allocated share of the cluster's resources and can dynamically scale their resource consumption up and down based on load. Figure 3.1 shows how two frameworks running on Mesos can share the clus-

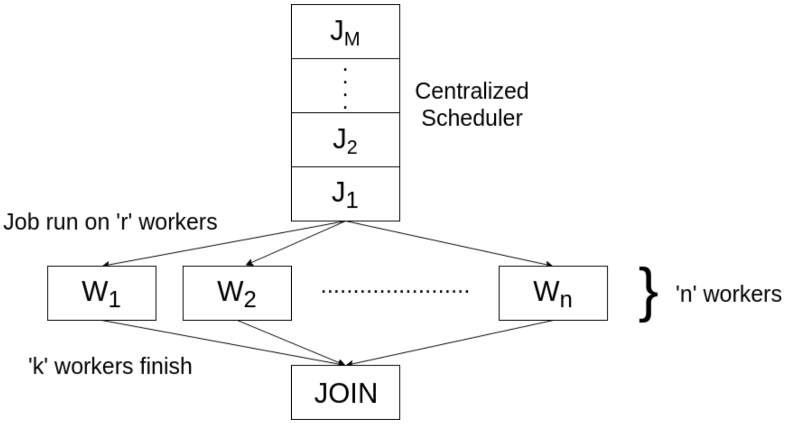

Figure 3.2: The (n,r,k) Fork-Join System with Centralized Scheduling.

ter's resources. Mesos communicates with a framework's scheduler using *callbacks*. It invokes different functions inside the scheduler where user defined behavior can be implemented for different callbacks. For example, the *resourceOffers* callback is invoked when Mesos wants to offer resources to the scheduler.

3.2 System Model

3.2.1 Fork-Join Systems

In *fork-join systems*, a job is forked into tasks which are run on statistically identical servers. Such a system may also implement redundancy so that all the forked tasks do not need to finish for the job to complete successfully. A general model for fork-join systems is the (n, r_f, r, k) fork-join model [24]. In this model, a job is forked on to r_f of n statistically identical servers. When any $r <= r_f$ tasks are ready to run on the servers, tasks are cancelled on individual servers so as to retain only r tasks. Of the r tasks that run, only k must finish for the job to complete successfully.

3.2.2 Credence as a Fork-Join System

Credence is modeled as a *(n,r,k)* fork-join system with *centralized scheduling*. The simple *(n,r,k)* fork-join system in Figure 3.2 shows jobs $J_1, J_2, ..., J_M$ waiting to be scheduled on the n workers. The (n, r, k) fork-join system is a special case of the (n, r_f, r, k) fork-join system with $r_f = r$. This means that any matrix multiplication operation, which we refer to as a *job*, is run on r of the n servers available, and only k of the servers must finish for the job to complete succesfully. Note that for coded matrix multiplication, k corresponds to the recovery threshold of the matrix multiplication operation, whereas for uncoded matrix multiplication, $k = r$. One can see that the fork-join model for Credence dovetails perfectly with the Mesos job and task model. In a cluster consisting of n nodes, the Credence centralized scheduler accepts r resource offers for a job from Mesos and runs matrix multiplication tasks on the r workers. Once k of the r workers finish, Credence can decode the result or save it for decoding offline. This corresponds to the $JOIN$ stage in Figure 3.2.

3.3 Architecture

3.3.1 Programming Model

Credence is built using the Dask [39] engine. Dask provides Credence with two important features: a DAG (Directed Acyclic Graph) scheduler and a distributed Numpy [48] array implementation. A developer implements a *driver program* to submit a job to Credence. Within the driver program, a developer can create distributed Dask arrays, encode them using the Generalized PolyDot codes, and multiply encoded or uncoded matrices. Encoded arrays are stored within a special data structure known as the *EncodedArray*. An EncodedArray holds no data by itself, but contains references to encoded Dask arrays on different workers. We will use the terms *driver program* and *job* interchangeable throughout the remainder of this thesis. Developers

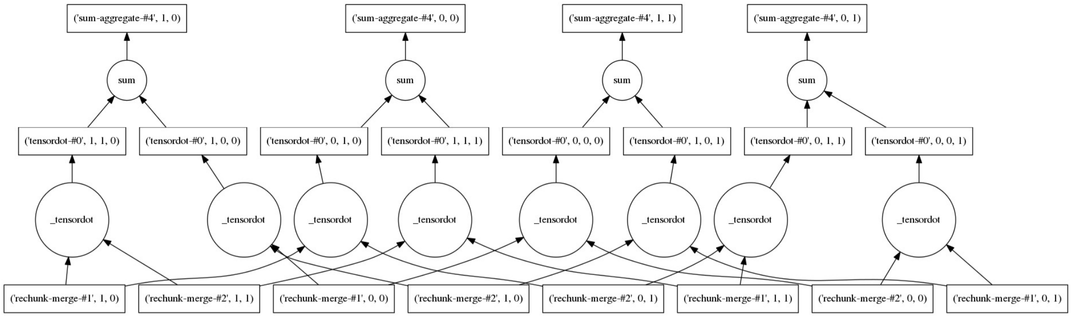
('sum-aggregate-#4', 1, 0)
('sum-aggregate-#4', 0, 0)
('sum-aggregate-#4', 1, 1)
('sum-aggregate-#4', 0, 1)
sum
sum
sum
sum
('tensordot-#0', 1, 1, 0)
('tensordot-#0', 1, 0, 0)
('tensordot-#0', 0, 1, 0)
('tensordot-#0', 1, 1, 1)
('tensordot-#0', 0, 0, 0)
('tensordot-#0', 1, 0, 1)
('tensordot-#0', 0, 1, 1)
('tensordot-#0', 0, 0, 1)
_tensordot
_tensordot
_tensordot
_tensordot
_tensordot
_tensordot
_tensordot
_tensordot
('rechunk-merge-#1', 1, 0)
('rechunk-merge-#2', 1, 1)
('rechunk-merge-#1', 0, 0)
('rechunk-merge-#2', 1, 0)
('rechunk-merge-#2', 0, 1)
('rechunk-merge-#1', 1, 1)
('rechunk-merge-#2', 0, 0)
('rechunk-merge-#1', 0, 1)

Figure 3.3: Multiplying Two Uncoded Matrices of Size (100x100) in Credence Using the Dask DAG Scheduler.

can also use Dask lazy operations to add tasks to a Dask computation graph and finally *compute* [39] the results.

Figure 3.3 shows the DAG constructed during the multiplication of two uncoded (100x100) matrices on 4 workers nodes on Credence. Rectangular nodes in the graph identify intermediate results while circular nodes identify the operations being performed on them. The $rechunk - merge$ nodes in the graph indicate the matrix number and the portion being multiplied. For example, $(rechunk - merge - \#1, 0, 0)$ identifies the submatrix at grid position $(0, 0)$ of the first matrix. Each *_tensordot* corresponds to the computation of one element $A_{ri}B_{ic}$ of the outer product for C_{rc} and each *sum* in the upper nodes of the graph correspond to the computation $C_{rc} = \sum A_{ri}B_{ic}$ as shown in Section 1.2.1.

Figure 3.4 shows the DAG generated during the multiplication of two (100x100) matrices encoded using MatDot codes with a polynomial of degree 2. (Note that Credence is able to support both MatDot codes and Polynomial codes using its Generalized PolyDot implementation). The path on the right side leading up to *solve* indicates the construction of the Vandermonde matrix used during interpolation, whereas the left side leading up to *solve* shows the construction of the matrix consisting of the MatDot encoding polynomial evaluated at three different points

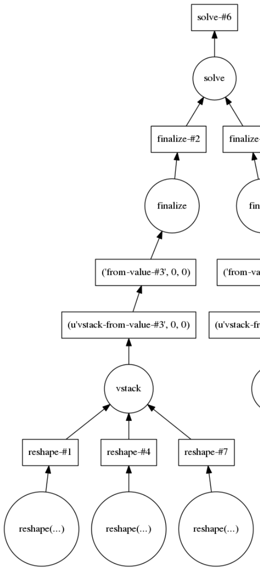

Figure 3.4: Multiplying Two Encoded Matrices of Size (100x100) in Credence Using the Dask DAG Scheduler.

by three different workers. As per the strategy laid out in [10], the matrices have been encoded beforehand. The *solve* operation indicates that the MatDot encoding polynomial is being interpolated to determine its coefficients. For this example, the second element in the resulting matrix is the product $C = AB$.

Credence currently supports only online decoding. This means results are decoded as soon as the recovery threshold for a matrix multiplication operation is met. Decoding can be distributed across the cluster or can be performed at the client node from where jobs are submitted. Distributed decoding is supported by the linear algebra primitives offered by Dask for its distributed Numpy arrays. Decoding at the client is done by Numpy's linear algebra library.

3.4 Mesos Schedulers in Credence

Every Credence job contains a *CredenceContext* instance using which it requests resources from the Credence scheduler and creates, encodes, and multiplies matrices on the cluster. The *CredenceContext* instance can interact with the following two kinds of schedulers.

3.4.1 Single Driver Scheduler

The *Single Driver Scheduler* is created and started by a driver program. It's sole purpose is to request resources from Mesos for the job and it terminates when the job completes. This scheduler is useful during debugging or testing driver programs, or if the developer knows that there is only one matrix multiplication job that is to be run on the cluster.

3.4.2 Configurable Scheduler

The *Configurable Scheduler* implements all of the Mesos scheduler callbacks plus a *get_driver_by_policy* function in which developers can define their own scheduling policies. Currently, only the *First-Come-First-Served (FCFS)* policy is supported. It is different from the Single Driver scheduler in that it is a long running daemon and does not terminate after the last job exits. This scheduler runs the following three remote object threads using which driver programs can communicate with it:

- *SchedulerInterface* : Driver programs make RPC calls on this interface to request resources from the Configurable Scheduler. Invocations to the SchedulerInterface contain the number of workers r required for the job , the number of CPUs, memory and disk required per worker, and optionally, the recovery threshold k for the job. The SchedulerInterface object then adds this information to a data structure inside the Configurable Scheduler. When Mesos offers resources to the Configurable Scheduler, the *get_driver_by_policy* logic dictates which driver inside the scheduler's data structure will be scheduled. In the current implementation of the Configurable Scheduler, invocations to the SchedulerInterface are *blocking.* This means that a driver program will block until *all* of the resources it requested are allocated to it and it is scheduled. The current implementation of the Credence Configurable Scheduler therefore supports only *gang scheduling*.

- *TerminateInterface* : Driver programs make RPC calls on this interface after their recovery threshold k is met. This allows the Configurable Scheduler terminate the Dask DAG scheduler for the job and clean up any driver related state it had maintained.
- *AddStatsInterface* : A Credence driver program can optionally use this interface to inform the scheduler about the time taken to encode and multiply matrices. This is explained in more detail in Section 4.

3.5 Local Linear Algebra in Credence

A primary objective behind Credence is to use standard linear algebra libraries for node local computations so that different coded matrix multiplication strategies can be compared on a standardized platform. In this regard, all of the linear algebra performed at each worker or at the master during polynomial interpolation are performed using *BLAS* [29]. The first BLAS library was written in FORTRAN in 1979, but there are a number of variants available today. Credence uses the *OpenBLAS* implementation of BLAS for linear algebra operations. For example, all matrix multiplications are performed using the *GEMM* kernel.

3.6 Evaluation

In this section, we compare the performance of coded matrix multiplication to uncoded matrix multiplication on Credence. Additionally, we also compare coded matrix multiplication on Credence to *MLLib*. MLLib is a machine learning library that

	1	2	3	4	5
Square	1000	5000	10000	25000	50000
Rectangular	1000x50	5000x100	10000x100	25000x1000	50000x2000
Matrix-Vector	1000	5000	10000	25000	50000

Table 3.1: Specifications of matrix multiplication experiments conducted on Credence. The first row shows the side of the square matrices being multiplied. For example, 1000 indicates that two square matrices of side 1000 are being multiplied. The second row shows the dimensions of rectangular matrices being multiplied with square matrices. For example, 1000x50 indicates that a square matrix of side 1000 is being multiplied with a rectangular matrix of dimensions 1000x50. The third row shows the length of the vector being multiplied with a square matrix of the same side. For example, 1000 indicates that a vector of length 1000 is being multiplied with a square matrix of side 1000. All the matrices and vectors we use are dense.

is built on Apache Spark [54] for distributed machine learning. It also exposes matrix abstractions of RDDs for linear algebra.

3.6.1 Methodology

To evaluate the performance of coded matrix multiplication on Credence to uncoded matrix multiplication, we multiply matrix with the following attributes on a cluster consisting 20, 50, 100, and 150 workers. We use two coded matrix mutliplication strategies in our experiments - MatDot codes and Polynomial Codes. The degree of the encoding polynomial for all experiments involving coded matrix multiplication using MatDot codes is 2 and for Polynomial codes is 9. Each matrix multiplication experiment is conducted in a different driver program, and the driver that is running the experiment is the only job running on Credence. We use the Single Driver Scheduler for all experiments and decode the results online.

We also evaluate the performance of running a large number of jobs on Credence. To do this, we use a randomly generated *trace file* consisting a jobs the system must run. The trace file is a *CSV* file where each line contains the specifications of the job to be run. The first column is the side of two square matrices that are to be multiplied, the second column identifies the matrix multiplication strategy to be used for the job, namely MatDot, Polynomial, or uncoded, and third column indicates the number of workers to use for the job. Again, we set the degree of degree of the encoding polynomial to 2 for MatDot codes and 9 for polynomial codes. The trace file we use for this experiment consists of 100 jobs. We use the Configurable Scheduler with FCFS scheduling for this experiment. We set the arrival rate λ of jobs to the scheduler to 2 per second.

All of our experiments are conducted in the AWS public cloud on *t2.xlarge* instances. *t2.xlarge* instances come with 4 Intel Xeon processors and 16 GB of memory.

A Chaos Engineering Approach To Testing Credence

We repeat the experiments listed above under an environment inspired by the *principles of chaos engineering* [1]. Chaos testing was popularized by Netflix and is based on the premise that modern distributed systems have scaled to an extent that they cannot reliably be tested by humans. Systems may therefore manifest unexpected or undesired behavior in production. Netflix built a set of tools called the *Simian Army*[1] to induce chaos in their production systems in an attempt to test their resilience. A commonly used open sourced tool in the Simian Army suite is *Chaos Monkey.* Chaos Monkey can be used to induce a number of types of failures in cloud based environments. For example, it can be configured to terminate virtual machines, generate computational load, corrupt disks and network packets, or kill processes. We use Chaos Monkey to simulate network latency, a common cause of straggling. Chaos Monkey simulates network latency using *tc*, a popular traffic shaping tool. *tc* adds latency to network traffic flowing through an interface by adding *qdisc*s, or queuing disciplines, in the kernel. All traffic that is outgoing on an interface is queued ac-

---

[1] The word *"simian"* identifies something "relating to, resembling, or affecting apes or monkeys". The idea behind the name *Simian Army* is that the Netflix engineers wanted their services to be available even if a monkey was destroying their datacenter.

cording to the queuing discipline configured. Chaos Monkey uses the default FIFO qdisc and adds latency that simulates WAN performance.

3.6.2 Results

In this section, we present the results of matrix multiplication on Credence and compare coded matrix multiplication to uncoded matrix multiplication on Credence and MLLib. Detailed results are shown in Appendix A. In this section, we analyze the results obtained and compare uncoded multiplication on Credence and MLLib with coded matrix multiplication on the basis of time taken per operation with and without simulated latency. We also run a trace of matrix multiplication jobs on Credence whle varying the number of available workers.

Without simulated latency, coded matrix multiplication is able to outperform uncoded matrix multiplication on MLLib and Credence in most cases. For small matrices being multiplied on a small number of workers (see Figure A1), the redundancy introduced by coded matrix multiplication adds a small overhead. In the worst case, coded matrix multiplication with polynomial codes is able to achieve 85% the speed taken by uncoded matrix multiplication.

A key point to note is that, contrary to expectations, uncoded matrix multiplication scales *negatively* with the number of workers. This means that for matrices of same sizes, adding more workers to the system will lead to a degraded performance. We attrbute this to the fact that blocked matrix multiplication in a distributed system incurs a massive communication cost as sub-matrices have to be transferred to more workers while computing the outer product at some grid position. SUMMA is able to overcome this because it assumes that the workers being utilized for the computation are physically arranged in an MPI grid. Owing to this fact, SUMMA is able to support targeted MPI broadcasts of sub-matrices and is therefore communication efficient. However, it is unreasonable to assume that a system will exhibit similar behavior in public cloud based environments where communication is done mostly over TCP/IP. Cloud providers provide no guarantee of whether the virtual machines requested will be on the same host or rack, much less be neatly arranged in an MPI grid.

We also observe the same behavior with uncoded blocked matrix multiplication on Spark's MLLib. Multiplication of dense blocked matrices in MLLib employs a *GridPartitioner* which creates an RDD [53] where keys are grid positions and values are sub-matrices. Computing outer products therefore requires a network intensive *shuffle* operation. The alternating least squares (ALS) example in the MLLib paper [35] uses a dense model and is tested only on a cluster of 30 nodes and performance on larger clusters is not analyzed. In [14], Gu et al. propose a strategy for dense matrix multiplication on MLLib that is able to achieve near linear scaling with cluster size. They use an adaptive algorithm that uses different matrix multiplication strategies based on the size of the matrices being multiplied. For example, if one of the matrices is smaller than a *broadcast threshold*, it is broadcasted to all the workers and the algorithm therefore does not incur the communication penalty associated with blocked matrix multiplication. If the dimensions of the submatrices are small or almost equal, they emply a traditional blocked matrix multiplication strategy. Otherwise, they employ the CARMA [5] strategy for matrix multiplication in distributed memory systems. This approach is useful because it does not build any functionality into Spark or MLLib, but is a library built on top of these systems. It would therefore be possible to implement a similar approach in Credence with minimal or no changes to underlying functionality. Coded matrix multiplication on the other hand has a constant communication cost in the sense that the amount of data communicated is dependent only on the recovery threshold of the operation and not the number of workers.

When the experiments are repeated under simulated latency conditions, the trends remain largely the same. The time taken by uncoded matrix multiplication on Spark and Credence approximately double and increase with cluster size. Coded matrix multiplication remains faster than uncoded matrix multiplication for two reasons, Firstly, as discussed above, the amount of communication does not depend on cluster size. Secondly, the performance of uncoded matrix multiplication is further degraded as transferring sub-matrices over a network that is affected by latency is slower.

We then test the scalability of Credence with number of jobs in the system. We expect the latency of the system for a given set of jobs to decrease as the number of workers increase. This is because the scheduler has more resources available to it and can schedule jobs in parallel on different workers. Figure A16 verifies this intuition. As we increase the number of workers, the time taken to run the same trace decreases almost linearly.

4

Towards Optimal Redundant Task Scheduling in Credence

In the previous chapter, we examined the architecture of Credence and compared it to an (n, r, k) fork-join system. In this chapter, we apply the analysis of fork-join systems in [24] to design an optimal redundancy policy for Credence to minimize job latency.

4.1 Optimal Choices for r in (n, r, k) Fork-Join Systems

In [24], Joshi et al. derive the bounds on latency and cost in (n, k) fork-join systems and propose a heuristic strategy to choosing the optimal amount of redundancy in (n, r_f, r, k) fork-join systems. Their results are useful because they assume that the worker nodes follow an arbitrary service time distribution. Assuming worker that all n workers are homogeneous, they claim that the optimal choices for r_f^* for r_f and r^* for r are given by:

$$r_f^* = r_{max} \tag{4.1}$$

$$r^* = argmin\ \hat{T}(r), \quad s.t.\ \hat{C}(r) \leqslant \gamma \tag{4.2}$$

where $\hat{T}(r)$ and $\hat{C(r)}$ are estimates of the expected latency $\mathbb{E}(T)$ and cost $\mathbb{E}(C)$, defined as follows:

$$\hat{T}(r) \triangleq \mathbb{E}[X_{k:r}] + \frac{\lambda r \mathbb{E}[X_{k:r}^2]}{2(n - \lambda r \mathbb{E}[X_{k:r}])} \tag{4.3}$$

$$\hat{C}(r) \triangleq r\mathbb{E}[X_{k:r}] \tag{4.4}$$

Here, r_{max} is a limit on the value of r_f so as to limit communication cost, γ is is constraint on the computing cost, and λ is the arrival rate of jobs in the system. Because Credence is an (n, r, k) fork-join system, $r_f = r$ for Credence and the problem of determining the optimal amount of redundant tasks for a reduces to finding a solution for Equation 4.2.

4.2 Design of a Heuristic Strategy for Optimal Redundancy in Credence

Recall from Section 3.4 the *AddStatsInterface* remote object in the Credence Configurable Scheduler. RPC calls on this object are a means by which driver programs can return information on the time taken to encode and multiply matrices by each of the r workers. Given the encoding time and multiplication times per worker for some job, the Configurable scheduler adds these values to get the total time taken by the worker for a job. It then adds these values to a collection that represents the distribution of worker service times for the environment Credence is running in. Given this worker service time distribution and the expected recovery threshold k, we can use Equation 4.2 to find the optimal r value for subsequent jobs. We assume that the service time distribution is stored as a histogram in memory on the Configurable Scheduler. Algorithm 2 shows how an optimal choice for r can be made in Credence. For simplicity, we assume no limit on γ.

Algorithm 2 Algorithm for Optimal Redundancy in Credence

```
procedure OPTIMAL_R(k,n)
    $r^* \leftarrow \varnothing$
    $\hat{T}^* \leftarrow \varnothing$
    for every r $\geqslant$ k and $\leqslant n$ do
        S $\leftarrow$ Sample $N$ values from service time distribution
        $\mathbb{E}[X_{k:r}] \leftarrow k^{th}$ smallest value in S
        Calculate $\hat{T}$(r) from Equation 4.2
        if $\hat{T}$(r) $<$ $\hat{T}^*$ then
            $\hat{T}^* \leftarrow \hat{T}(r)$
            $r^* \leftarrow$ r
    return $r^*$
```

Note that the steps in the algorithm above can be taken at the scheduler without adding excessive overheads as Algorithm 2 runs in linear time with complexity $O(n-k)$ if N, the number of samples drawn from the service time distribution, is a constant. Note also that we assume that the job size, or the sizes of the matrices being multiplied, is constant. If the job size varies, the service time distribution increases in dimensionality.

4.3 Results

To analyze the performance of the heuristic strategy for optimal redundancy, we compare the time taken by the scheduler to run 40 jobs with optimal values for r chosen by Algorithm 2 with the time taken to run the same 40 jobs for sub-optimal values of r. In our experiments, we run jobs with $k = 3$ on a cluster of 20 *t2.xlarge* instances with FCFS scheduling. We experiment with different arrival rates and matrix sizes. All experiments are run after the scheduler has been able to collect 3000 readings for the service time distribution. Detailed results are presented in Appendix A. For jobs with optimal values of r, speedups as high as 3x are observed. An interesting point to note is the latency for higher values of λ is lower than that of lower values. This shows that Credence is able to cope with rapidly arriving jobs.

5

Conclusion

This thesis is focused on the applications of coded computing and the design and evaluation of a system that uses the Generalized PolyDot matrix multiplication strategy. In Chapter 2, we examine how coded computing can be applied to real problems in machine learning. In Chapter 3, we study the design of Credence, a system that uses coded matrix multiplication. In Chapter 4, we apply apply the analysis of (n, r, k) fork-join systems in [24] to choosing optimal redundancy in Credence. The remainder of this chapter is organized as follows. We first discuss the insights learned from this thesis and then mention some open problems which we leave to future work.

5.1 Lessons Learned

In Section 2, we saw that coded matrix multiplication can be applied to a variety of machine learning and scientific computing [23] problems. Note that if problems are formulated intelligently, as in Section 2, the overhead of encoding matrices can be avoided at every iteration. The examples introduced in Section 2 fall under the category of *model parallel* machine learning techniques. Model parallel techniques are characterized by models that are so large that they cannot be held in memory.

This is in contrast to *data parallel* machine learning, where models are replicated across nodes and are trained in parallel from different samples of training data.

In Section 3, we introduce the Credence platform for coded computing. We find that coded matrix multiplication in Credence leads to predictable performance when WAN level latencies are simulated in the network. Another point to note is that blocked matrix multiplication is difficult to scale owing to the communication overhead. Apart from Marlin [14], we find that this issue is not sufficiently covered in literature. The broadcast based multiplication strategy, while communication efficient, is difficult to scale. For example, consider the training of a DNN as introduced in Section 2. Broadcasting a model after updation to every worker at every iteration would be a very expensive operation. Broadcasting very large models, such as the Netflix prize matrix [27], would be next to impossible. Moreover, broadcast based multiplication assumes that one of the matrices can be held in memory at every worker.

In Section 4, we leverage the ideas behind distribution based scheduling to choosing optimal redundancy in Credence. We find that choosing an optimal number of redundant tasks can lead to a 3x improvement in latency for a trace of jobs. Because the strategy makes no assumption of worker service times, we anticipate that it can be extended to a variety of environments.

5.2 Open Problems

5.2.1 Numerical Errors in Matrix Decoding

Notice that the decoding operation in coded matrix multiplication using Generalized PolyDot codes involves solving the linear system $Ax = B$ where each element in B is the result of multiplying the results of encoding polynomial for each matrix at at different points v_i, each element in x_i in x is a matrix that is the coefficient of v^i in the encoding polynomial, and A is the Vandermonde matrix in which the knots are each v_i at which the encoding polynomials are evaluated. As shown in [37], Vandermonde matrices are ill conditioned except when the knots v_i are spaced on or about the circle $C(0, 1 = [x : |x| = 1])$. We employ this strategy in Credence, but have observed that the condition number for the matrix A exceeds 10^{15}. for encoding polynomials that have degree 12 or more. This makes inverting the matrix A without significant numerical errors difficult. A possible solution could be to choose v_i from the set of Chebyshev points:

$$x_k = \cos \frac{k-1}{2\pi} \pi, \quad k = 1, 2, ..., n \tag{5.1}$$

Polynomial interpolation with Chebyshev points is known to have lower errors than other points [?].

Bibliography

[1] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering." *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[2] J. Dean, "Software engineering advice from building large-scale distributed systems," *CS295 Lecture at Stanford University*, 2007.

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

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on.* IEEE, 2013, pp. 261–272.

[6] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *Advances In Neural Information Processing Systems (NIPS)*, 2016, pp. 2092–2100.

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

[8] S. Dutta, Z. Bai, T. M. Low, and P. Grover, "Codenet: Training Large Neural Networks in presence of Soft-Errors," 2018.

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

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

[11] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia, "Reducing latency via redundant requests: Exact analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 347–360, 2015.

[12] A. Geist, "How to kill a supercomputer: Dirty power, cosmic rays, and bad solder," *IEEE Spectrum*, vol. 10, 2016.

[13] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *Nsdi*, vol. 11, no. 2011, 2011, pp. 24–24.

[14] R. Gu, Y. Tang, Z. Wang, S. Wang, X. Yin, C. Yuan, and Y. Huang, "Efficient large scale distributed matrix computation with spark," in *Big Data (Big Data), 2015 IEEE International Conference on.* IEEE, 2015, pp. 2327–2336.

[15] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing.* IEEE Computer Society, 2010, pp. 691–696.

[16] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml," in *Proceedings of the Seventh ACM Symposium on Cloud Computing.* ACM, 2016, pp. 98–111.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[18] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High Performance Computing.* Springer, 2015.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[20] K. H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[21] 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.

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

[23] H. Jeong, T. M. Low, and P. Grover, "Coded fft and its communication overhead," *arXiv preprint arXiv:1805.09891*, 2018.

[24] G. Joshi, E. Soljanin, and G. Wornell, "Efficient redundancy techniques for latency reduction in cloud systems," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 2, no. 2, p. 12, 2017.

[25] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler Mitigation in Distributed Optimization through Data Encoding," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 5440–5448.

[26] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

[27] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[29] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[30] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," in *IEEE International Symposium on Information Theory (ISIT)*, 2016, pp. 1143–1147.

[31] K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Coded computation for multicore setups," in *IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2413–2417.

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

[33] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing.* ACM, 2014, pp. 1–14.

[34] S. Li, M. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A Fundamental Tradeoff Between Computation and Communication in Distributed Computing," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, 2018.

[35] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[36] M. Miranda, "The threat of semiconductor variability: As transistors shrink, the problem of chip variability grows," *IEEE Spectrum*, vol. 28, 2012.

[37] V. Y. Pan, "How bad are vandermonde matrices?" *SIAM Journal on Matrix Analysis and Applications*, vol. 37, no. 2, pp. 676–694, 2016.

[38] S. Rendle, D. Fetterly, E. J. Shekita, and B.-y. Su, "Robust large-scale machine learning in the cloud," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 2016, pp. 1125–1134.

[39] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, no. 130-136. Citeseer, 2015.

[40] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[41] N. R. Shanbhag, S. Mitra, G. de Veciana, M. Orshansky, R. Marculescu, J. Roychowdhury, D. Jones, and J. M. Rabaey, "The search for alternative computational paradigms," *IEEE Design & Test of Computers*, vol. 25, no. 4, 2008.

[42] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *12th USENIX Symposium on Networked Systems Design and Implementation.* USENIX Association, 2015, pp. 513–527.

[43] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, “Deepface: Closing the gap to human-level performance in face verification,” in *IEEE conference on computer vision and pattern recognition*, 2014, pp. 1701–1708.

[44] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, “Gradient coding,” in *Machine Learning Systems Workshop, Advances in Neural Information Processing Systems (NIPS)*, 2016.

[45] R. A. Van De Geijn and J. Watts, “Summa: Scalable universal matrix multiplication algorithm,” *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[46] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, p. 5.

[47] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, “Low latency via redundancy,” in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies.* ACM, 2013, pp. 283–294.

[48] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, “The numpy array: a structure for efficient numerical computation,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[49] D. Wang, G. Joshi, and G. Wornell, “Efficient task replication for fast response times in parallel computation,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 599–600.

[50] Y. Yang, P. Grover, and S. Kar, “Computing Linear Transformations With Unreliable Components,” *IEEE Transactions on Information Theory*, vol. 63, no. 6, 2017.

[51] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, “Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication,” in *Advances In Neural Information Processing Systems (NIPS)*, 2017, pp. 4403–4413.

[52] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, “Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding,” *arXiv preprint arXiv:1801.07487*, 2018.

[53] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[54] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[55] C. Zhao, X. Bai, and S. Dey, "Evaluating transient error effects in digital nanometer circuits," *IEEE Transactions on Reliability*, vol. 56, no. 3, pp. 381–391, 2007.

Appendix A

Experimental Results

In this appendix, we list some of the results explained in Section 3.6. Figures A1 through A15 depict the time taken for each experiment on 20, 50, 100, and 150 workers. The numbers $(1, 2, 1)$ and $(2, 2, 2)$ indicate the number of horizontal partitions m of the first matrix, the number of vertical partitions n of the first matrix (also the number of horizontal partitions of the second matrix), and the number of vertical partitions d of the second matrix as shown in Section 1.2.2. $(1, 2, 1)$ is therefore MatDot codes with an encoding polynomial of degree 2 whereas $(2, 2, 2)$ is Polynomial codes with an encoding polynomial of degree 9. Figure A16 shows the scalability of Credence with respect to the number of jobs in the system. Figures A17 through A19 compare the run times for a trace consisting of 40 identical jobs when the values of r are chosen using Algorithm 2 with sub-optimal values of r.

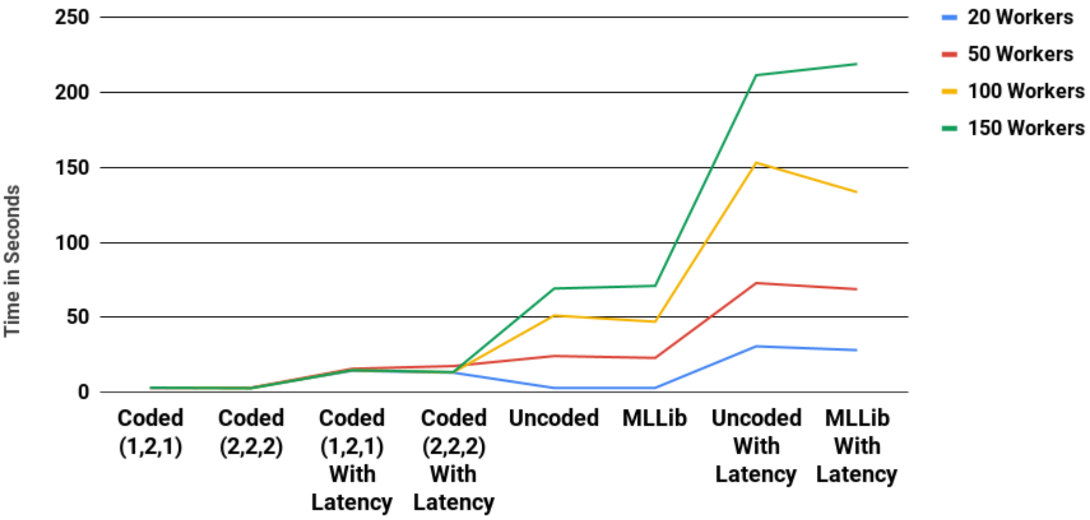

Figure A1: Multiplying Two 1000x1000 Square Matrices on 20, 50, 100, and 150 Workers using Credence.

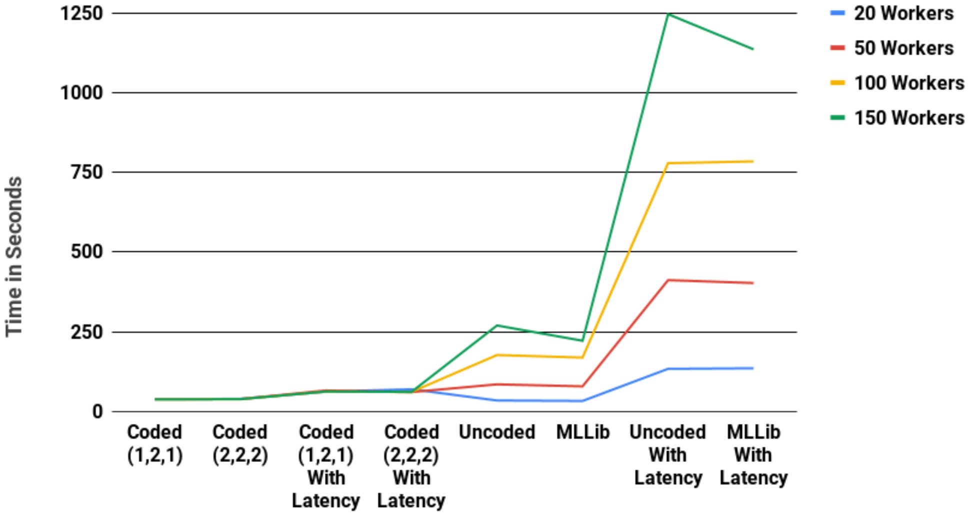

Figure A2: Multiplying Two 5000x5000 Square Matrices on 20, 50, 100, and 150 Workers using Credence.

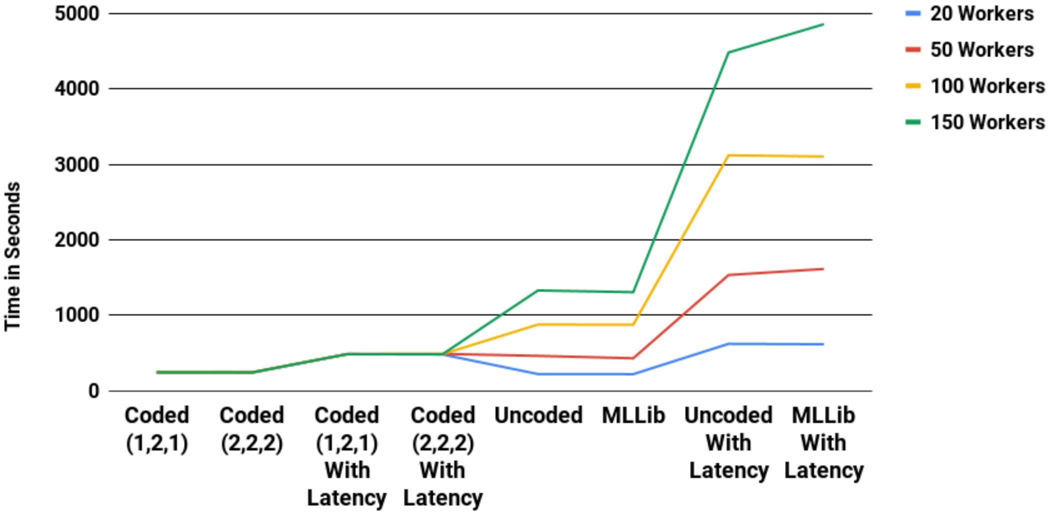

Figure A3: Multiplying Two 10000x10000 Square Matrices on 20, 50, 100, and 150 Workers using Credence.

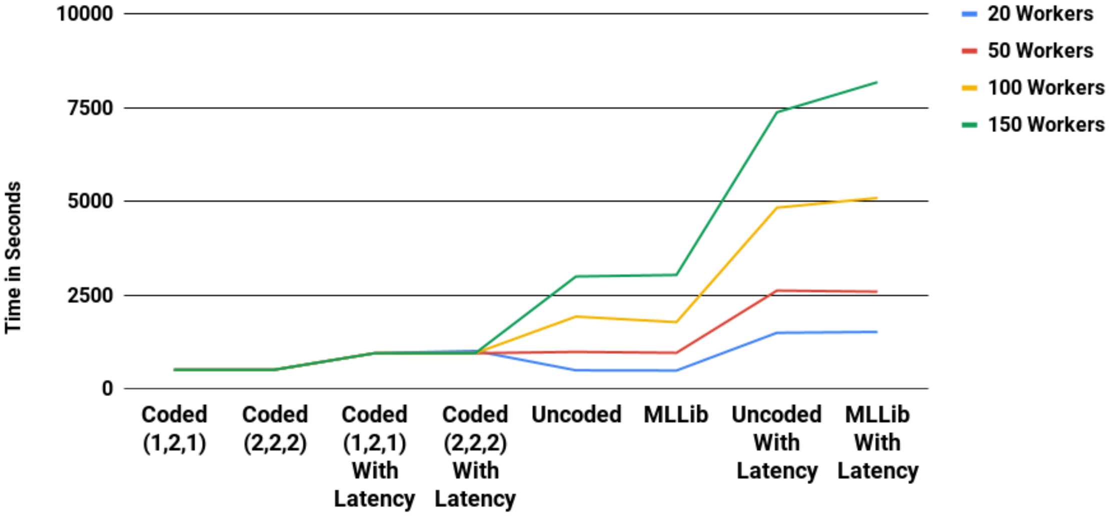

Figure A4: Multiplying Two 25000x25000 Square Matrices on 20, 50, 100, and 150 Workers using Credence.

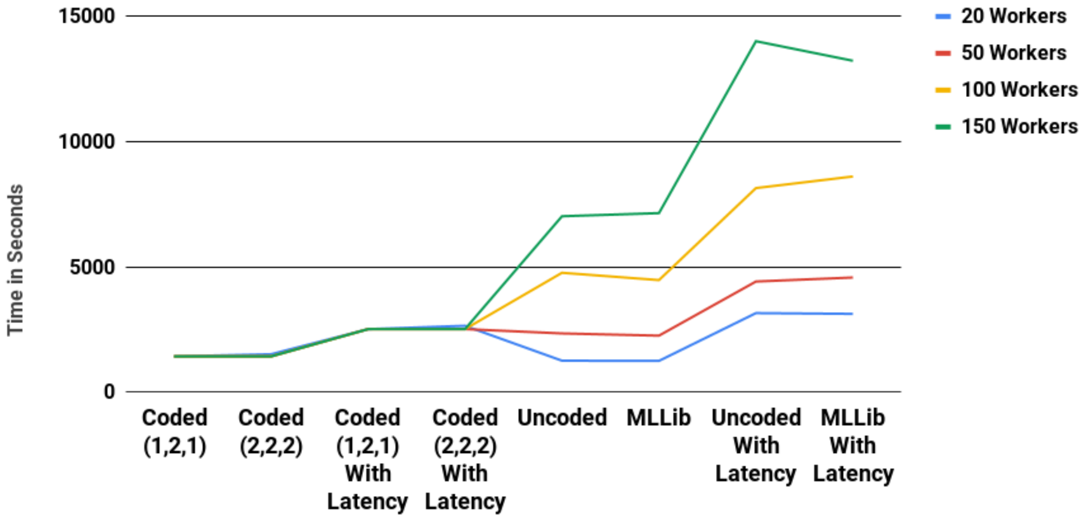

Figure A5: Multiplying Two 50000x50000 Square Matrices on 20, 50, 100, and 150 Workers using Credence.

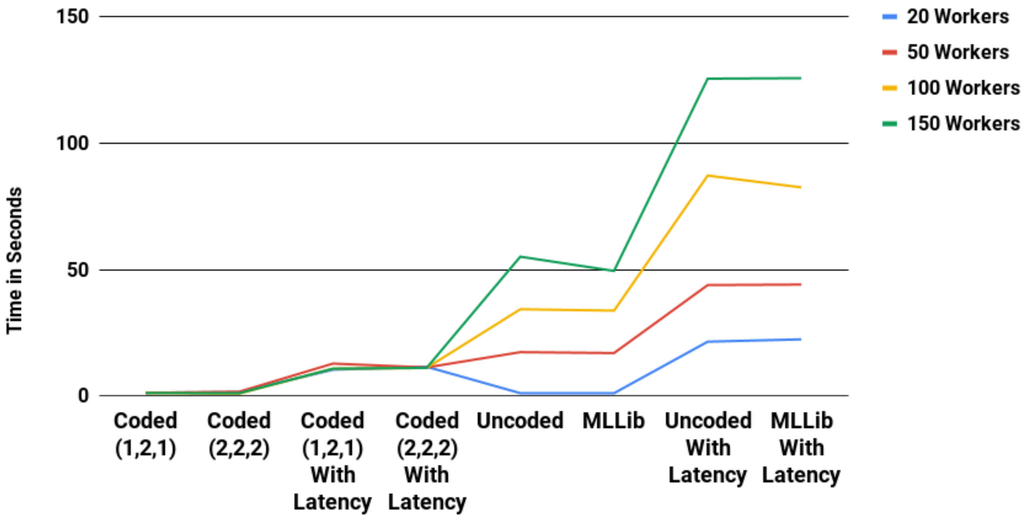

Figure A6: Multiplying a 1000x1000 Square Matrix with a 1000x50 Rectangular Matrix on 20, 50, 100, and 150 Workers using Credence.

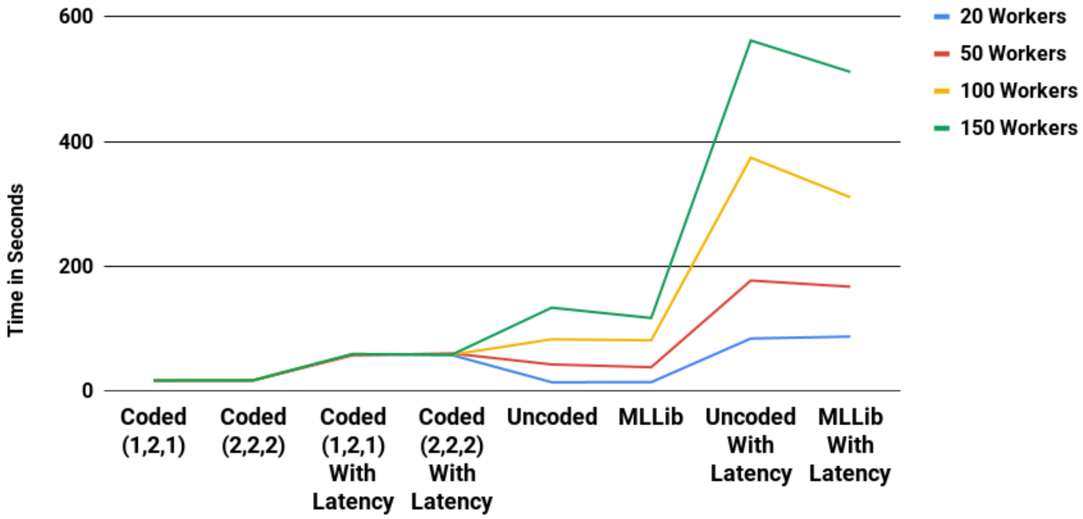

Figure A7: Multiplying a 5000x5000 Square Matrix with a 5000x100 Rectangular Matrix on 20, 50, 100, and 150 Workers using Credence.

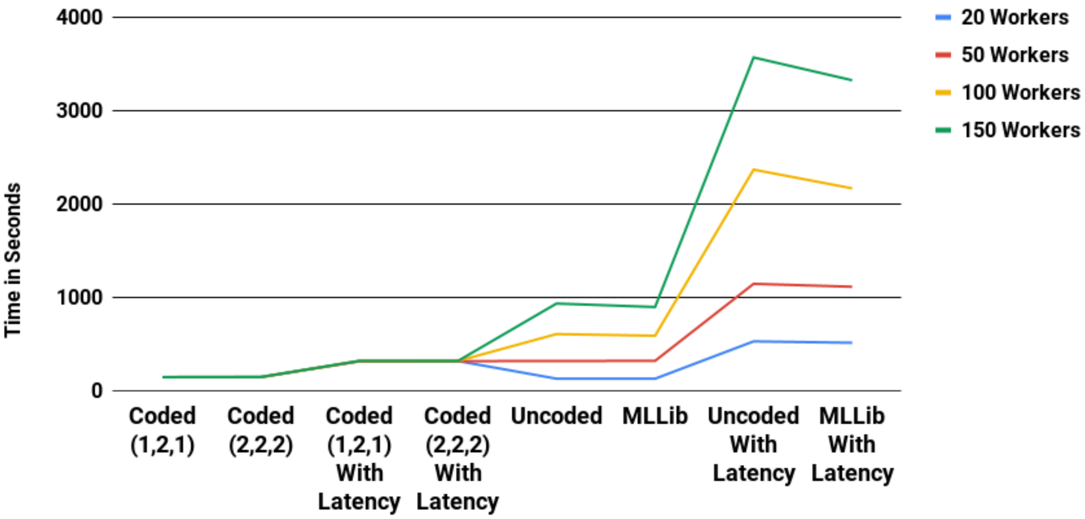

Figure A8: Multiplying a 10000x10000 Square Matrix with a 10000x100 Rectangular Matrix on 20, 50, 100, and 150 Workers using Credence.

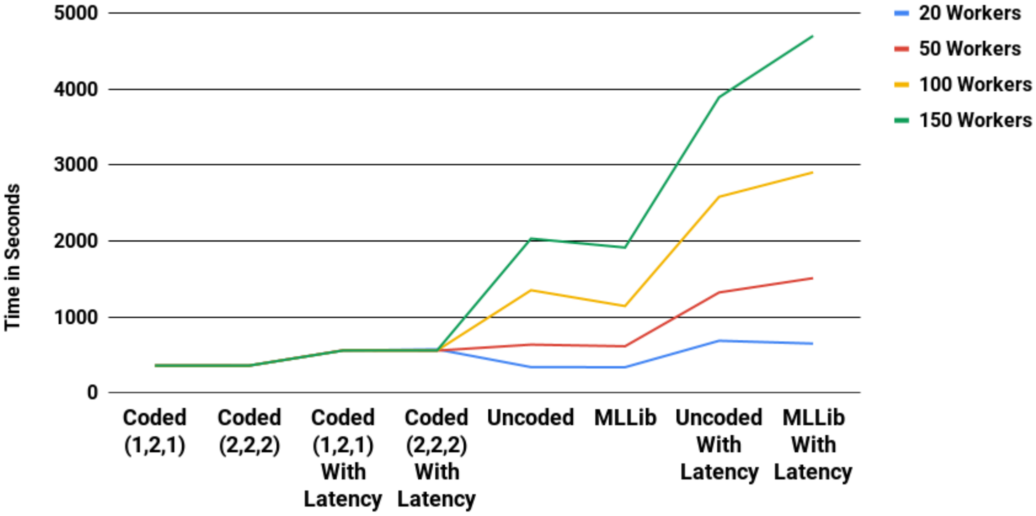

Figure A9: Multiplying a 25000x25000 Square Matrix with a 25000x1000 Rectangular Matrix on 20, 50, 100, and 150 Workers using Credence.

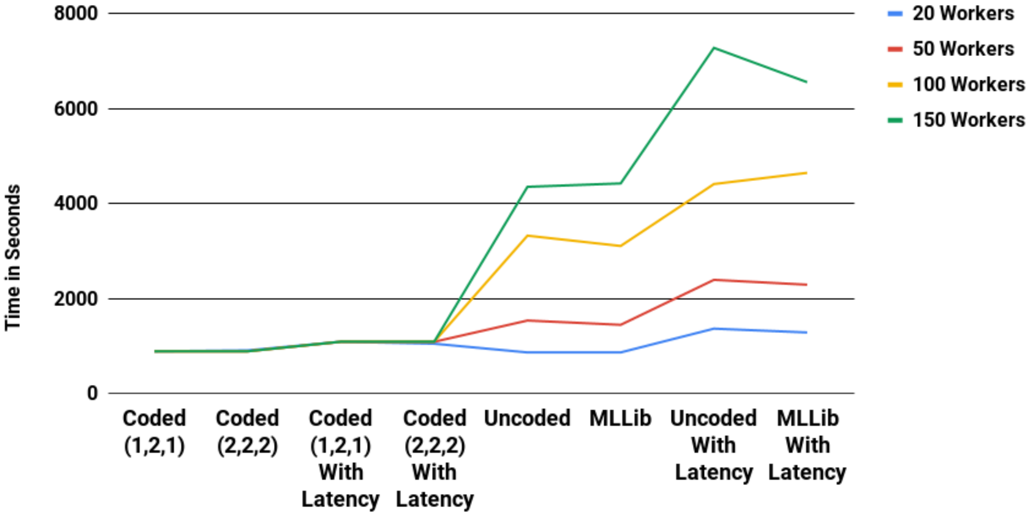

Figure A10: Multiplying a 50000x50000 Square Matrix with a 50000x2000 Rectangular Matrix on 20, 50, 100, and 150 Workers using Credence.

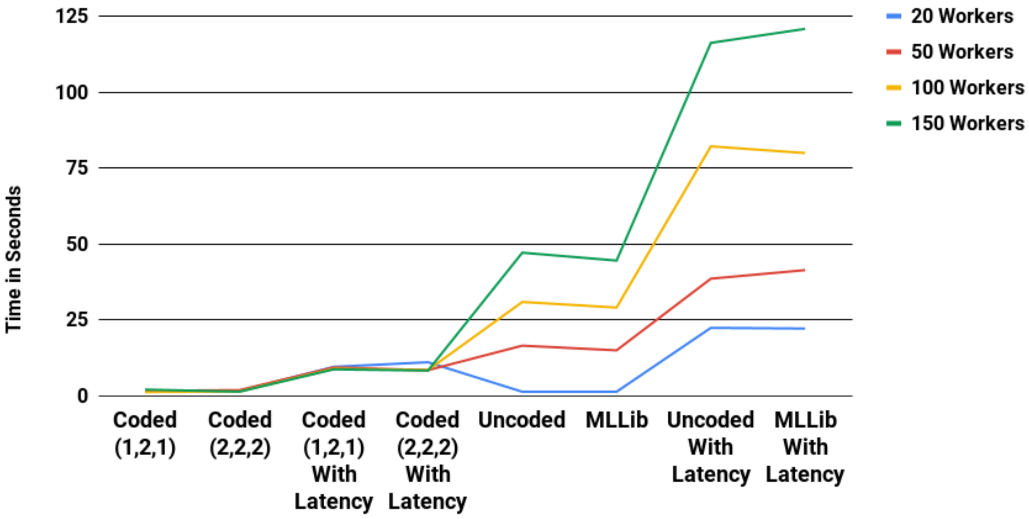

Figure A11: Multiplying a 1000x1000 Square Matrix with a Vector of Length 1000 on 20, 50, 100, and 150 Workers using Credence.

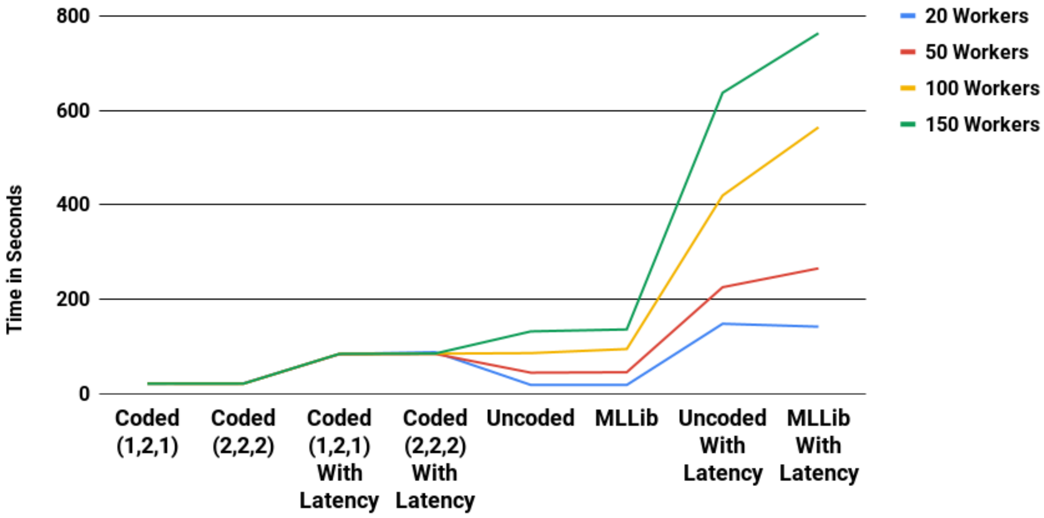

Figure A12: Multiplying a 5000x5000 Square Matrix with a Vector of Length 5000 on 20, 50, 100, and 150 Workers using Credence.

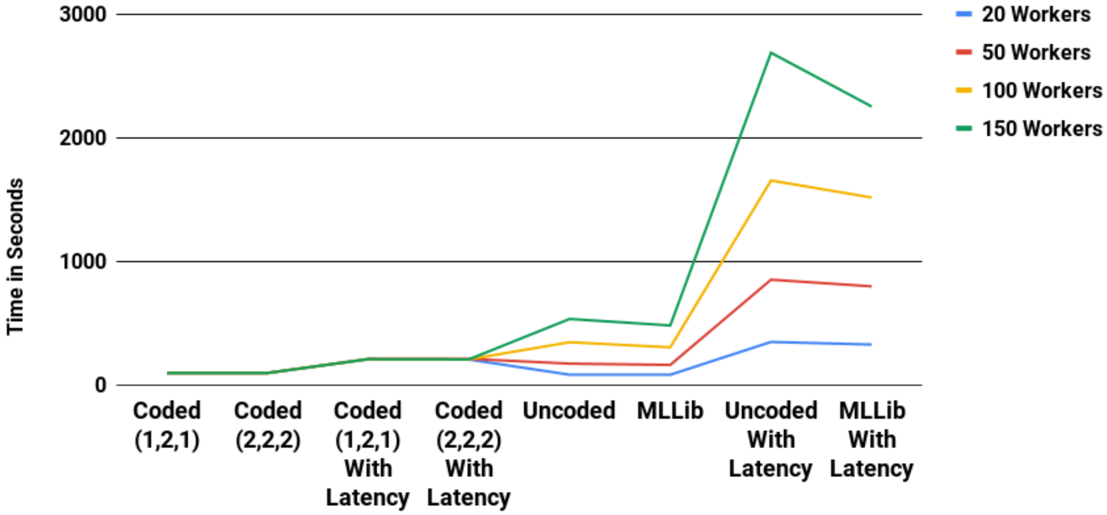

Figure A13: Multiplying a 10000x10000 Square Matrix with a Vector of Length 10000 on 20, 50, 100, and 150 Workers using Credence.

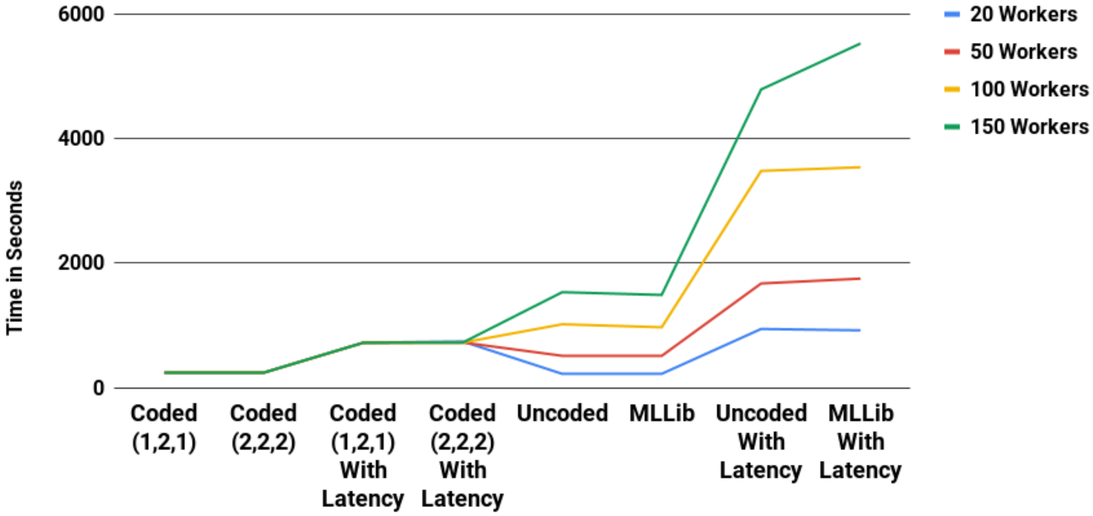

Figure A14: Multiplying a 25000x25000 Square Matrix with a Vector of Length 25000 on 20, 50, 100, and 150 Workers using Credence.

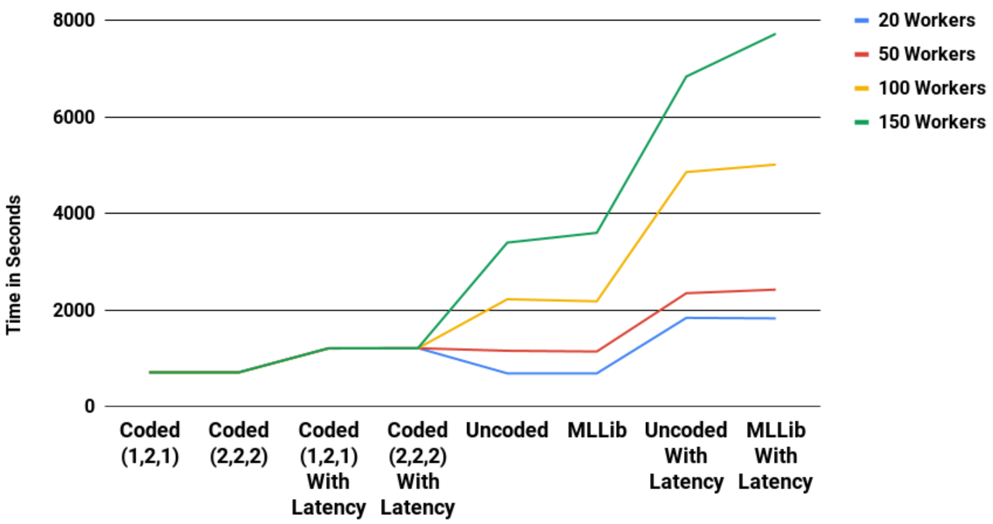

Figure A15: Multiplying a 50000x50000 Square Matrix with a Vector of Length 50000 on 20, 50, 100, and 150 Workers using Credence.

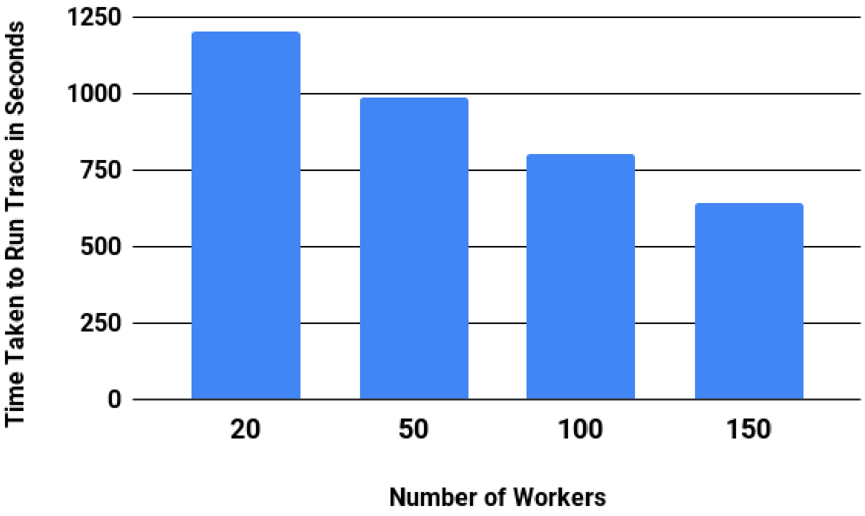

Figure A16: Running a Trace of Matrix Multiplication Jobs on Credence

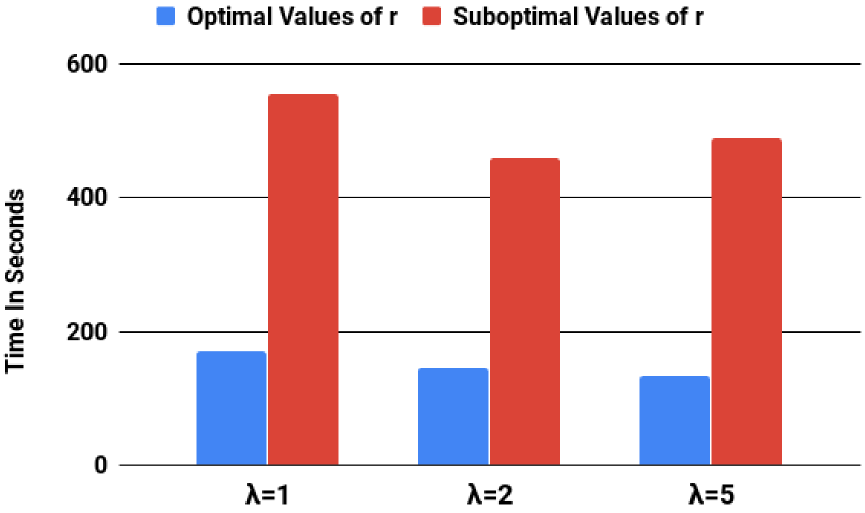

Figure A17: Runtimes for Optimal Values of r vs Sub-Optimal Values of r While Multiplying Two 1000x1000 Square Matrices

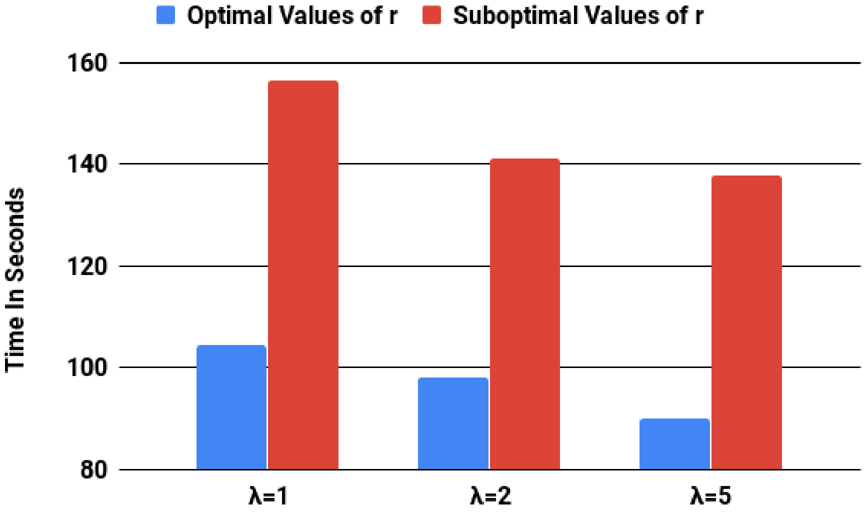

Figure A18: Runtimes for Optimal Values of r vs Sub-Optimal Values of r While Multiplying A 1000x1000 Square Matrix with a 1000x50 Rectangular Matrix

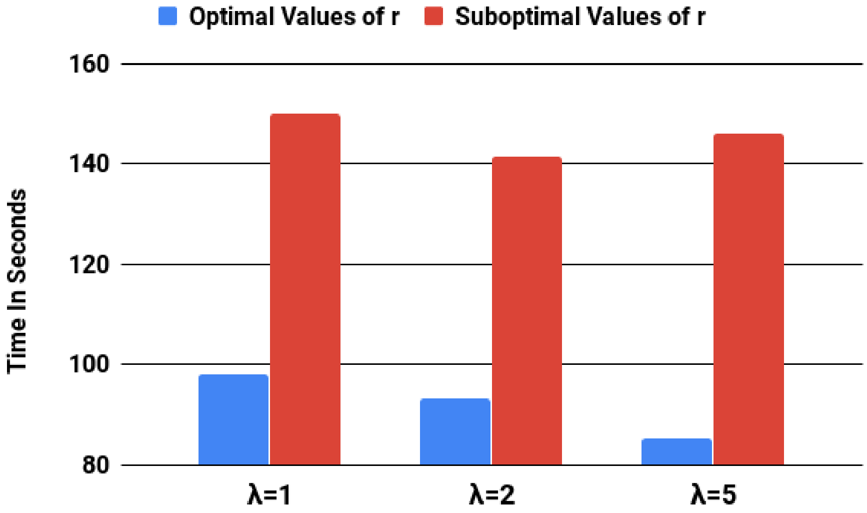

Figure A19: Runtimes for Optimal Values of r vs Sub-Optimal Values of r While Multiplying A 1000x1000 Square Matrix with a Vector of Length 1000