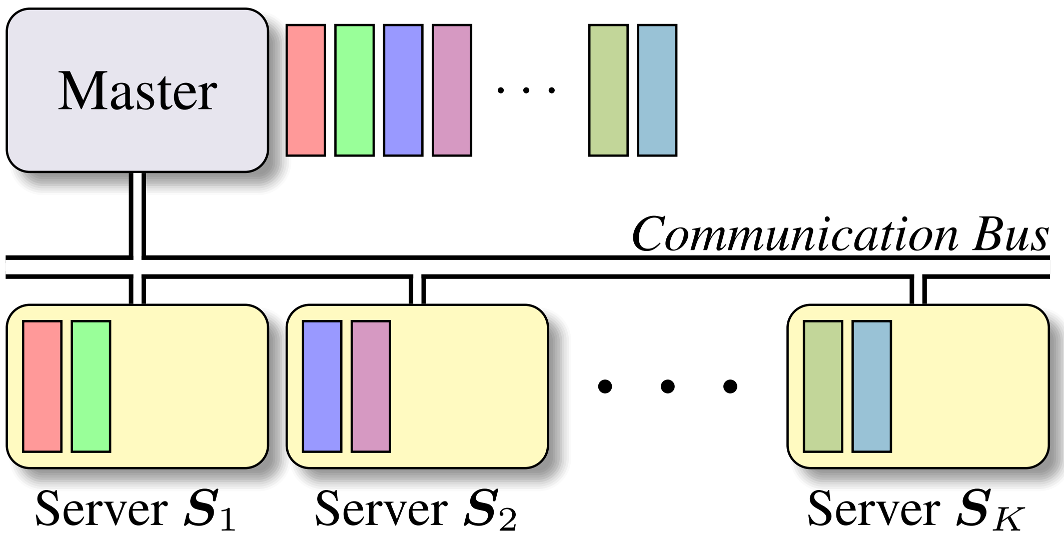
CHALMERS
UNIVERSITY OF TECHNOLOGY



Coding for Distributed Computing

Investigating and improving upon coding theoretical frameworks for distributed computing

Master's thesis in Communication Engineering

ALBIN SEVERINSON

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

MASTER'S THESIS EX014/2017

Coding for Distributed Computing

Investigating and improving upon coding theoretical frameworks for distributed computing

ALBIN SEVERINSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Communication Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Coding for Distributed Computing
Investigating and improving upon coding theoretical frameworks for distributed computing
ALBIN SEVERINSON

Supervisor and examiner:
Alexandre Graell i Amat, Department of Electrical Engineering

Co-supervisor:
Eirik Rosnes, Simula@UiB, Bergen, Norway

Master's Thesis EX014/2017
Department of Electrical Engineering
Division of Communication Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of the considered distributed computing system model.

Typeset in LaTeX
Gothenburg, Sweden 2017

Coding for Distributed Computing
Investigating and improving upon coding theoretical frameworks for distributed computing
ALBIN SEVERINSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Distributed computing has emerged as an effective way of tackling increasingly complex computational problems. However, distributed computing systems bring significant challenges. Among them, the problems of straggling servers and bandwidth scarcity have recently received significant attention. The straggler problem is a synchronization problem characterized by the fact that a distributed computing task must wait for the slowest server to complete its computation. On the other hand, distributed computing tasks typically require that data is moved between servers during the computation, the so-called *data shuffling*, which is a challenge in bandwidth-constrained networks.

We consider the distributed computing task of multiplying a set of vectors with a matrix. This operation is a key component of machine learning and several other data-intensive applications. For this scenario, coding theoretical solutions have been proposed for both the straggler and data shuffling problem by Lee *et al.* and Li *et al.* respectively. Furthermore, Li *et al.* recently unified these ideas in a common framework and showed a fundamental tradeoff between computational delay and communication load. This coding framework is based on maximum distance separable (MDS) codes of code length proportional to the number of rows of the matrix, which can be very large.

We propose a block-diagonal coding scheme consisting of partitioning the matrix into submatrices and encoding each submatrix using a shorter MDS code. We show that the assignment of coded matrix rows to servers to minimize the communication load can be formulated as an integer program with a nonlinear cost function, and propose an algorithm to solve it. We further prove that, up to a level of partitioning, the proposed scheme does not incur any loss in terms of computational delay (as defined by Li *et al.*) and communication load compared to the scheme by Li *et al.*. We also show numerically that, when the decoding time is also taken into account, the proposed scheme significantly lowers the overall computational delay with respect to the scheme by Li *et al.*. For heavy partitioning, this is achieved at the expense of a slight increase in the communication load.

Keywords: coding, information, theory, distributed computing, machine learning

Acknowledgements

I would first and foremost like to thank my two supervisors Alexandre Graell i Amat and Eirik Rosnes. I have enjoyed working with you greatly and I have learned a tremendous amount from you along the way. I am also grateful for my partner Naemi Jönsson and my family, who have supported me through the series of ups and downs that is research. I would also like to thank my friends from the master's thesis room for making the last year a lot more enjoyable!

Albin Severinson, Gothenburg, June 2017

Contents

List of Figures **xi**

List of Acronyms **xii**

List of Symbols **xiii**

1 Introduction **1**
- 1.1 In This Thesis . . . 2
- 1.2 Thesis Outline . . . 3

2 Preliminaries **5**
- 2.1 Erasure Correcting Codes . . . 5
- 2.2 Distributed Computing . . . 6
- 2.3 Probabilistic Runtime Model . . . 7
- 2.4 Bandwidth Scarcity . . . 8
- 2.5 The Straggler Problem . . . 9
- 2.6 The Unified Scheme . . . 11

3 Coding for Distributed Computing **13**
- 3.1 Distributed Computing Model . . . 13
  - 3.1.1 Map Phase . . . 13
  - 3.1.2 Shuffle Phase . . . 14
  - 3.1.3 Reduce Phase . . . 16
- 3.2 Coded Computing Schemes . . . 16
  - 3.2.1 Uncoded Computing . . . 16
  - 3.2.2 Coded MapReduce . . . 17
  - 3.2.3 Straggler Coding . . . 17
  - 3.2.4 The Unified Scheme . . . 17

4 Block-Diagonal Coding **19**
- 4.1 Assignment of Coded Rows to Batches . . . 19
- 4.2 Performance of the Block-Diagonal Coding . . . 21
  - 4.2.1 Communication Load . . . 22
  - 4.2.2 Computational Delay . . . 23
- 4.3 Assignment Solvers . . . 24
  - 4.3.1 Heuristic Solver . . . 25
  - 4.3.2 Branch-and-Bound Solver . . . 26

4.3.2.1 Branch 26
4.3.2.2 Bound 26
4.3.3 Hybrid Solver 26

5 Numerical Results **27**
5.1 Coded Computing Comparison 27
5.2 Assignment Solver Comparison 28

6 Conclusions and Future Work **31**

Bibliography **33**

List of Figures

2.1 **a:** Standard MapReduce with six messages sent in total. **b:** Coded MapReduce with only two messages sent. 8

2.2 The effect of straggling servers. 9

2.3 Coded distributed matrix multiplication. The results returned from any two servers are sufficient to decode the overall computation output. 10

2.4 **a:** Server subtask runtime distribution. **b:** Total computation runtime distribution. 10

3.1 Servers (yellow boxes) finish their respective subtasks in random order. 14

4.1 BDC scheme for $T = 3$ partitions. 19

4.2 Storage design for $m = 20$, $N = 4$, $K = 6$, $q = 4$, $\mu = 1/2$, and $T = 5$. 20

4.3 Multicasting coded values between servers S_1, S_2, and S_3. 21

4.4 Reduce delay of the unified and SC schemes; normalized by that of our proposed scheme. **(a)** As a function of partitioning for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. **(b)** As a function of system size for $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$. 24

4.5 Assignment of coded rows to servers. 25

5.1 **(a)** The tradeoff between partitioning and performance for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. **(b)** Performance dependence on system size for $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$. 28

5.2 **(a)** Solver performance as a function of partitioning for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. **(b)** Performance dependence on system size for $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$. 29

List of Acronyms

BDC	block-diagonal coding
CMR	coded MapReduce
MDS	maximum distance separable
RS	Reed-Solomon
SC	straggler coding
UC	uncoded computing
WSC	warehouse-scale computer

List of Symbols

$\boldsymbol{A}$	input matrix
m	number of rows of $\boldsymbol{A}$
n	number of columns of $\boldsymbol{A}$
$\boldsymbol{\Psi}$	encoding matrix
$\boldsymbol{C}$	coded matrix
r	number of rows of $\boldsymbol{C}$
$\boldsymbol{x}$	input vector
$\boldsymbol{y}$	output vector
N	number of input/output vectors
K	number of servers
μ	fraction of matrix rows stored at each server
B	batch
$\mathcal{T}$	batch label
S	server
$\mathcal{S}$	set of servers
q	required map phase subtask results for an MDS erasure code
g	required map phase subtask results for a general erasure code
σ	computational complexity
$F_{(g)}$	runtime of the g-th fastest server
$\mathcal{Q}$	set of the first q servers to complete their subtasks
$\mathcal{G}$	set of the first g servers to complete their subtasks
D_{map}	computational delay of the map phase
D_{reduce}	computational delay of the reduce phase
D	overall computational delay
L	communication load
$\mathcal{Z}_j^{(S)}$	set of intermediate values computed by server S for $\boldsymbol{x}_j$
$\mathcal{W}_S$	set of indices of the vectors server S is responsible for
$\mathcal{V}_{\mathcal{S}}^{(S)}$	set of values needed by S and known exclusively by servers in $\mathcal{S}$
T	number of partitions
$\boldsymbol{\psi}$	$r/T \times m/T$ MDS encoding matrix
$\boldsymbol{c}_i^{(t)}$	i-th coded row within partition t
$\boldsymbol{P}$	assignment matrix
$U_{\mathcal{Q}}^{(S)}$	number of remaining needed values by S in $\mathcal{Q}$
$U_{\mathcal{Q}}$	total number of remaining needed values by servers in $\mathcal{Q}$
$\mathbb{Q}^q$	superset of all sets $\mathcal{Q}$
$L_{\mathbb{Q}}$	communication load of the unicast phase for the set $\mathcal{Q}$

1
Introduction

We are increasingly dependent on cloud services such as those offered by Google, Amazon, Facebook, and Microsoft. The computing capacity required to run these services is increasing proportionally. However, computer processor cores are no longer becoming significantly faster. Instead, more of them are put into the same machine [1]. This approach has allowed building more powerful computers. However, software that can effectively utilize several cores is often much more complex than its single-core counterpart [2, Chapter 1]. Furthermore, this statement no longer captures the full complexity of modern computing. Even by building computers with multi-core processors, computing performance is no longer increasing fast enough. To keep up with demand, building massive distributed computing clusters has emerged as one of the most effective ways of tackling increasingly complex computation problems [1,3–5]. Theses clusters, referred to as “warehouse-scale computers” (WSCs) [3], may be composed of thousands of servers.

WSCs are not just data centers. Traditional data centers typically host a large number of relatively small computing systems, each running a separate application. Each computing system may be architecturally different, composed of very different hardware, and may not communicate with other systems in the same data center. In contrast, a WSC is a single computing system composed of a very large number of relatively homogeneous hardware and software components. Furthermore, a WSC runs only a small number of very large applications that may require a significant amount of communication between servers. WSCs currently power the services offered by companies such as Google, Amazon, Facebook, and Microsoft [3].

Achieving high availability (Internet applications typically aim for at least 99.99% uptime, i.e., about an hour of downtime per year [3]) and efficiency for applications running on WSCs is a major challenge due to the large number of components that may malfunction. Even if ensuring fault-free operation in a system composed of 10000 servers is possible, it would surely be extremely expensive [3]. Therefore, WSC applications must be designed to gracefully tolerate large numbers of both transient and permanent component errors with little or no impact on its availability and/or performance. These errors can be considered a kind of system noise. Furthermore, building more robust applications allows building WSCs out of cheaper (and thus more unreliable) components. Tools from information and coding theory (the science of information and how to represent it) have been very successful in dealing with noise and errors in other engineering contexts. For example, codes are a critical part of modern cellular communication systems and are increasingly used in distributed storage systems. It is our belief that coding will become a key component of future computing systems as well.

To effectively utilize the resources of WSCs, several distributed computing frameworks have been proposed. These frameworks provide a layer of abstraction, thus allowing a programmer to write applications for WSCs without having to consider its full complexity. In particular, MapReduce [6] has gained significant attention as a means of effectively utilizing large computing clusters. For example, Google routinely performs computations over several thousands of servers using MapReduce [6]. One of the main application areas is machine learning and data analytics. MapReduce implementations include means of compensating for component failures, for example by detecting them and rescheduling the failed tasks on other servers. Among the challenges present in WSCs, the problems of straggling servers and bandwidth scarcity have recently received significant attention. The straggler problem is a synchronization problem characterized by the fact that a distributed computing task must wait for the slowest server to complete its computation. On the other hand, distributed computing tasks typically require that data is moved between servers during the computation, commonly referred to as *data shuffling*, which is a challenge in bandwidth-constrained networks.

One of the key operations in distributed computing is that of multiplying a matrix with a set of vectors. We refer to this as the distributed matrix multiplication problem. This operation is a key component of machine learning and several other data-intensive applications. For this scenario, coding theoretical solutions have been suggested for both the data shuffling and straggler problem by Li *et al.* [7] and Lee *et al.* [8], respectively. In [7], a structure of repeated computation tasks across servers was proposed, enabling coded multicast opportunities that significantly reduce the required bandwidth to shuffle the results. In [8], the authors showed that codes can be applied to a linear computation task (e.g., multiplying a vector with a matrix) to alleviate the effects of straggling servers and reduce the computational delay. For the scheme in [8], the output of the computation is encoded and must be decoded to produce the final result. The decoding is a form of post-processing.

Furthermore, in [9] a unified coding framework was presented and a fundamental tradeoff between computational delay and communication load was identified. The ideas of [7, 8] can be seen as particular instances of the framework in [9], corresponding to the minimization of the communication load or the computational delay. However, the code proposed in [9] is complex to decode since the code length is proportional to the number of rows of the matrix to be multiplied, which may be very large. We show that the scheme in [9] may slow down rather than speed up the overall computation due to its high decoding complexity.

1.1 In This Thesis

In this thesis, we introduce a novel encoding scheme for the distributed computing task of multiplying a matrix $\boldsymbol{A}$ with a set of vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$. The proposed encoding is equivalent to partitioning the matrix and applying smaller (and thus much less complex) codes to each submatrix separately. The storage design for the proposed block-diagonal encoding can be cast as an integer optimization problem with a nonlinear objective function, whose computation scales exponentially with the problem size. We propose a heuristic solver for efficiently solving the optimization problem, and a branch-and-bound approach for improving on the resulting solution iteratively. We exploit a dynamic programming approach to speed up the branch-and-bound operations. Furthermore, we prove that up to a certain partitioning level, partitioning does not increase the computational delay (as defined in [9]) and the communication load with respect to the scheme in [9]. Interestingly, when the decoding time is taken into account, the proposed scheme achieves an overall computational delay significantly lower than the one of the scheme in [9]. This is due to the fact that the proposed scheme allows using significantly shorter codes, hence reducing the decoding complexity and the decoding time. Also, numerical results show that a high level of partitioning can be applied at the expense of only a slight increase in the communication load.

The results presented in this thesis have resulted in a paper submitted to the 2017 IEEE Information Theory Workshop [10].

1.2 Thesis Outline

This remainder of this thesis is organized as follows. Chapter 2 covers some preliminaries. In Chapter 3, we present the distributed matrix multiplication problem and system model in detail. We also define the coded schemes of [7–9] in terms of our system model. In Chapter 4, we propose a novel block-diagonal coding (BDC) scheme for the bandwidth scarcity and straggler problem. In the same chapter we present the performance analysis of the proposed scheme. We also present the storage design optimization formulation and propose algorithms to solve it. In Chapter 5, we present numerical results comparing the performance of our proposed scheme to that of the schemes in [7–9]. Finally, Chapter 6 draws some conclusions and makes suggestions for future work.

2 Preliminaries

In this chapter, we give a brief introduction of the main ideas used in this thesis. First, we introduce erasure correcting codes. Next, we give an overview of distributed computing and introduce the probabilistic runtime model used in this thesis. Finally, we explain the bandwidth scarcity and straggler problem in distributed computing. We also describe the coded computing schemes introduced in [7–9]. These concepts are the cornerstones of this thesis.

2.1 Erasure Correcting Codes

Erasure correcting coding is the procedure of adding redundancy to a piece of information that may be partially erased during transmission or storage. The information is commonly referred to as a message that is transmitted over a noisy channel. The code makes it possible to recover (decode) the original information even if only parts of the message are received (i.e., erasures occurred during transmission). Two common scenarios are packet loss during transmission between computers and hard disk drive failures in storage systems (data storage can be considered transmission over time). An erasure correcting code encodes a message of m symbols (represented by a sequence of bits in computers) into a longer message of r coded symbols such that the original m message symbols can be decoded from a subset of the r coded symbols.[1] Repetition codes and maximum distance separable (MDS) codes are two of the most important types of erasure correcting codes. Reed-Solomon (RS) codes are a well known class of MDS codes.

Repetition codes encode a message by repeating it. Specifically, a repetition code simply repeats each message symbol r times. The message can then be recovered from any one of its r replicas. Repetition codes are widely used due to their simplicity. For example, a backup hard disk drive is a repetition code.

MDS codes exploit a structure of linear dependence between the encoded symbols. Specifically, a message of m symbols encoded with an (r, m) MDS code results in r coded symbols, any m of which is sufficient to decode the original m message symbols. As an example, consider a message consisting of two numbers A and B. The message can be encoded using a $(3, 2)$ MDS code by producing the coded message A, B, and $A + B$. The original message symbols can be decoded from any two coded symbols. MDS codes extend to any numbers m and $r \geq m$. The encoding

[1]The number of source and coded symbols are usually denoted by k and n, respectively. We denote these by m and r instead for consistency with the rest of the thesis.

and messages can be represented as matrices $\boldsymbol{\Psi}$ and $\boldsymbol{A}$. The matrix representation of the coded message $\boldsymbol{C}$ is then obtained as $\boldsymbol{C} = \boldsymbol{\Psi}\boldsymbol{A}$.

2.2 Distributed Computing

Distributed computing is a method of making software applications that can execute several computing tasks concurrently. In this thesis, we refer to distributed computing systems as systems composed of discrete computing units (servers) that communicate by exchanging messages. For example, a WSC [3] is a distributed computing system. An application running on a WSC can execute computing tasks on several servers concurrently and the tasks can communicate by sending messages over the shared network. Distributed computing systems also allow utilizing resources at geographically separate locations. However, building applications that can efficiently utilize potentially thousands of servers is challenging [2, Chapter 1].

In response, computing frameworks such as Apache Spark [11] and MapReduce [6] have been proposed. These frameworks provide a layer of abstraction that allows programmers to more easily write applications that can utilize large numbers of servers concurrently. By handling most of the complexity inherent to distributed computing within the framework, the application can be made much simpler. For example, by detecting and compensating for component failure within the framework, the application can be written as if it is executed within a fault-free system.

MapReduce has become one of the most popular frameworks due to its simplicity and ability to scale to very large numbers of servers. We base the system model of this thesis on this framework. A computation within the MapReduce framework is composed of three distinct phases.

1. **Map:** A distributed mapping of input values to intermediate values. More precisely, each server performs some computation on the input values it has been assigned to produce a (potentially different) number of intermediate values.[2]

2. **Shuffle:** Moving intermediate values between servers by sending messages over the network, i.e., data shuffling.

3. **Reduce:** Reducing the intermediate values to the final computation output in a distributed fashion. Specifically, each server computes some number of output values from the set of intermediate values it has either computed locally or received in the shuffle phase.

MapReduce applications are created by writing the map and reduce phase computations. The computing tasks are then automatically assigned to servers by the framework and executed with no intervention by the programmer. Improvements to the MapReduce framework can thus improve the performance of MapReduce applications without any change to the application itself.

[2]The mapping may in some cases be random. However, this is unusual [6] and we will assume that the map phase is deterministic.

MapReduce may be deployed on the same servers as a distributed storage system [6]. In this case, input values may be mapped to intermediate values by the same server that stores those values. Furthermore, the computation output can be stored in the distributed storage system by simply leaving the data stored on the server that reduced it (perhaps also sending it to other servers for redundancy). We give an example of how distributed matrix multiplication can be performed within the MapReduce framework in Section 2.4.

2.3 Probabilistic Runtime Model

We adopt the probabilistic model of the computation runtime of [8]. We assume that running a computation on a single server takes a random amount of time according to the shifted-exponential cumulative probability distribution

$$F(t) = \begin{cases} 1 - \mathrm{e}^{-\left(\frac{t}{\sigma} - 1\right)}, & \text{for } t \geq \sigma \\ 0, & \text{otherwise} \end{cases},$$

where σ is the number of multiplications and divisions required to complete the computation. We do not take addition and subtraction into account as those operations are orders of magnitude faster [12]. Furthermore, we refer to the parameter σ associated with some operation as its computational complexity. For example, the complexity of computing the inner product of two length-n vectors is $\sigma = n$ as it requires performing n multiplications. The shift of the shifted exponential should be interpreted as the minimum amount of time the computation can be completed in. The distribution tail accounts for transient disturbances such as transmission and queuing delays. The complexity of an operation σ affects both the shift and the tail of the distribution. Furthermore, the tail of the distribution is the cause of the straggler problem.

When the algorithm is split into K parallel subtasks that are run across K servers, we denote the runtime of the subtask running on server S_k by F_k. As in [8], we assume that $F_1, \dots, F_K$ are independent and identically distributed random variables with distribution $F(Kt)$. We denote by $F_{(g)}, g = 1, \dots, K$, the g-th order statistic, i.e., the g-th smallest variable of $F_1, \dots, F_K$. The runtime of the g-th, $g = 1, \dots, K$, fastest server to complete its subtask is thus given by $F_{(g)}$. $F_{(g)}$ is a Gamma distributed random variable [13] with expectation and variance given by [14]

$$f(\sigma, K, g) \triangleq \mathbb{E}\left(F_{(g)}\right) = \sigma \left(1 + \sum_{j=K-g+1}^{K} \frac{1}{j}\right), \tag{2.1}$$

$$\mathrm{Var}\left(F_{(g)}\right) = \sigma^2 \sum_{j=K-g+1}^{K} \frac{1}{j^2}.$$

For a system with K servers performing a computation of complexity σ we denote by $f(\sigma, K, g)$ the expected runtime of the g-th fastest server $F_{(g)}$.

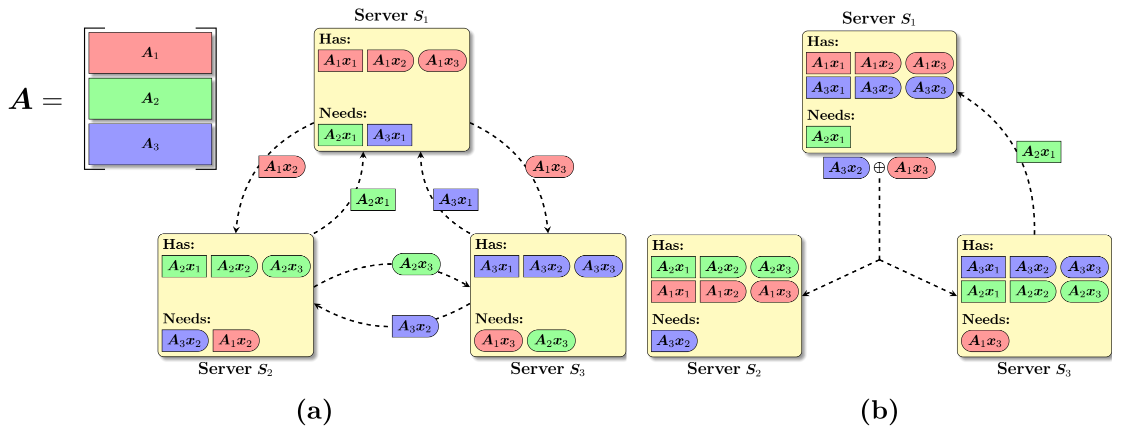


Figure 2.1: a: Standard MapReduce with six messages sent in total. **b:** Coded MapReduce with only two messages sent.

2.4 Bandwidth Scarcity

In this section, we explain the bandwidth scarcity problem and the solution proposed by Li *et al.* [7] in more detail. Many distributed computing applications require data to be shuffled among servers. Furthermore, computing clusters are in some cases limited by the available network bandwidth rather than by the available computing resources. For example, 50% to 70% of the overall runtime of distributed computing tasks is spent on data shuffling in some cases [15]. For this scenario, Li *et al.* [7] proposed a strategy of trading increased computational delay for a lower communication load. Specifically, they proposed performing repeated computations to create coded multicasting opportunities. A coded multicast is a message sent to several servers simultaneously such that all recipients can cancel parts of the message and thus recover the data intended for that specific server.

The scheme proposed in [7] extends the MapReduce [6] framework and is referred to as coded MapReduce. We illustrate both the standard and coded MapReduce for the task of multiplying three input vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$ with a matrix $\boldsymbol{A}$ to produce three output vectors $\boldsymbol{y}_1 = \boldsymbol{A}\boldsymbol{x}_1, \boldsymbol{y}_2 = \boldsymbol{A}\boldsymbol{x}_2, \boldsymbol{y}_3 = \boldsymbol{A}\boldsymbol{x}_3$ in Fig. 2.1. We perform the computation using 3 servers S_1, S_2, S_3 and we assume that the input vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$ are known to all servers at the start of the computation. We require that server S_i, $i = 1, 2, 3$, stores the output vector $\boldsymbol{y}_i$ after the MapReduce computation is completed. We first cover the standard MapReduce computation and then explain the extension proposed in [7]. First, $\boldsymbol{A}$ is split into three submatrices $\boldsymbol{A}_1$, $\boldsymbol{A}_2$, $\boldsymbol{A}_3$, and $\boldsymbol{A}_i$ is assigned to server S_i. These are the map phase input values. Next, server S_i maps the input value $\boldsymbol{A}_i$ to the three intermediate values $\boldsymbol{A}_i\boldsymbol{x}_1$, $\boldsymbol{A}_i\boldsymbol{x}_2$, $\boldsymbol{A}_i\boldsymbol{x}_3$, i.e., the map computation performed by each server consists of multiplying the submatrix it has been assigned with the 3 input vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$. In the shuffle phase, intermediate values are transmitted over the network such that server S_i stores the intermediate values $\boldsymbol{A}_1\boldsymbol{x}_i$, $\boldsymbol{A}_2\boldsymbol{x}_i$, $\boldsymbol{A}_3\boldsymbol{x}_i$. In Fig. 2.1, the values stored by each server after the map phase, i.e., the output of the map phase computation performed by that server, are written under the "Has" heading. The values it

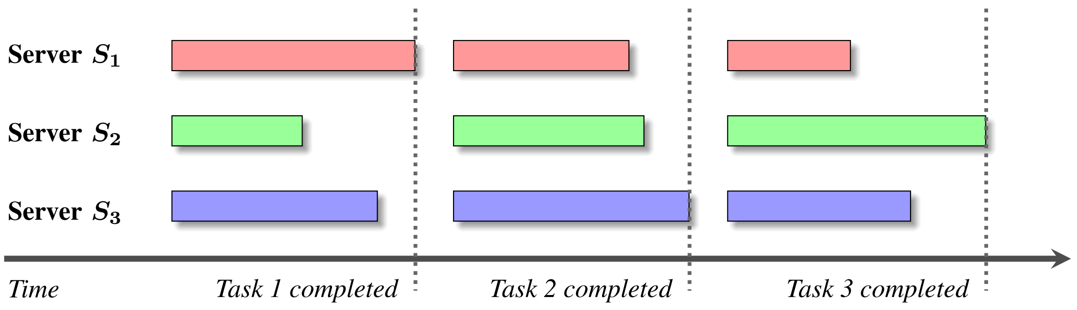


Figure 2.2: The effect of straggling servers.

needs from the other servers are written under the “Needs” heading. In this case, 6 messages have to be sent in total. Finally, in the reduce phase server S_i concatenates the 3 intermediate values $\boldsymbol{A}_1\boldsymbol{x}_i$, $\boldsymbol{A}_2\boldsymbol{x}_i$, $\boldsymbol{A}_3\boldsymbol{x}_i$ to obtain the output vector $\boldsymbol{y}_i$.

We now explain how coded MapReduce differs. For the coded MapReduce computation, each submatrix is assigned to a unique set of 2 servers. Specifically, $\boldsymbol{A}_1$ is assigned to S_1, S_2, $\boldsymbol{A}_2$ is assigned to S_2, S_3, and $\boldsymbol{A}_3$ is assigned to S_1, S_3. The map phase is performed in the same way as for standard MapReduce, i.e., each server multiplies the submatrices it has been assigned with the input vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$. After the map phase, each server needs only 1 intermediate value from the other servers as it has computed 2 values locally. Furthermore, in the shuffle phase one server can provide the data needed by the two other servers by multicasting a single coded message to both servers simultaneously. For example, server S_1 can provide the data needed by both servers S_2 and S_3 by multicasting $\boldsymbol{A}_3\boldsymbol{x}_2 \oplus \boldsymbol{A}_1\boldsymbol{x}_3$, where $\oplus$ is the bitwise XOR operation. As each recipient has computed locally one of the values, it can cancel it from the message and recover the value it needs. Either server S_2 or S_3 can then simply transmit $\boldsymbol{A}_2\boldsymbol{x}_1$ to S_1. Using this coded approach, only a total of 2 messages have to be transmitted. The reduce phase is unchanged from standard MapReduce. In this instance, each server has to do twice the work. However, only a third as many messages have to be transmitted over the network. If the system performance is limited by the available bandwidth, this may be a worthwhile tradeoff. In fact, Li *et al.* showed that this strategy can speed up some bandwidth-limited computations by a factor of 1.97 to 3.39 [16]. This concept is generalized in Chapter 3.

2.5 The Straggler Problem

Distributed computing tasks must typically wait for the slowest server to complete its subtask. This problem is commonly referred to as the straggler problem and is one of the main bottlenecks in distributed systems [8]. We illustrate the effect of straggling servers in Fig. 2.2. Servers may straggle for a variety of reasons such as hard disk issues slowing down read/write performance, or by being temporarily overloaded [6]. Repetition codes, i.e., assigning the same computing task to several servers have long been used to deal with stragglers [6]. For example, queries to Google services may be simultaneously sent to several servers. Only the result

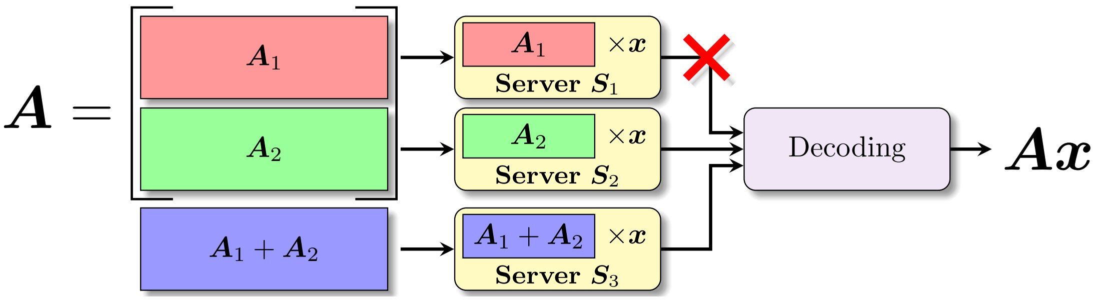

$\boldsymbol{A} =$
$\boldsymbol{A}_1$
$\boldsymbol{A}_2$
$\boldsymbol{A}_1 + \boldsymbol{A}_2$
$\boldsymbol{A}_1 \times \boldsymbol{x}$
Server $\boldsymbol{S}_1$
$\boldsymbol{A}_2 \times \boldsymbol{x}$
Server $\boldsymbol{S}_2$
$\boldsymbol{A}_1 + \boldsymbol{A}_2 \times \boldsymbol{x}$
Server $\boldsymbol{S}_3$
Decoding
$\boldsymbol{Ax}$

Figure 2.3: Coded distributed matrix multiplication. The results returned from any two servers are sufficient to decode the overall computation output.

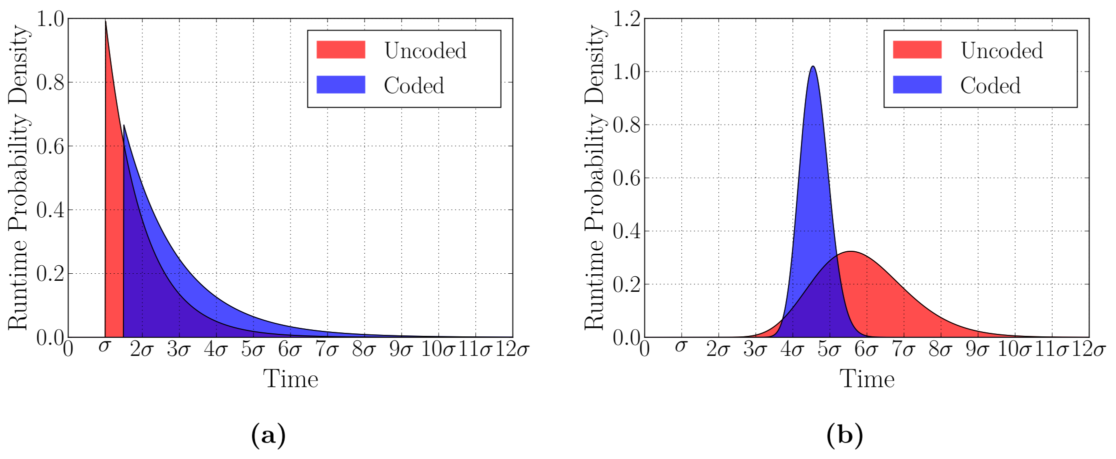


Figure 2.4: **a:** Server subtask runtime distribution. **b:** Total computation runtime distribution.

returned by the fastest server is shown to the user [17]. This approach was shown to speed up computations by close to 70% in [6]. However, Lee *et al.* showed that distributed linear computations, such as matrix vector multiplication, could be sped up further by using MDS codes rather than repetition codes [8].

We illustrate the scheme proposed in [8] for the task of multiplying a matrix $\boldsymbol{A}$ with a vector $\boldsymbol{x}$ using 3 servers in Fig. 2.3. The matrix $\boldsymbol{A}$ is split into two submatrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ that are encoded using a $(3, 2)$ MDS code, thus creating an additional parity submatrix $\boldsymbol{A}_1 + \boldsymbol{A}_2$. One submatrix is assigned to each server. Clearly, the computation output $\boldsymbol{Ax}$ can be decoded from any two of the results $\boldsymbol{A}_1\boldsymbol{x}$, $\boldsymbol{A}_2\boldsymbol{x}$, $(\boldsymbol{A}_1 + \boldsymbol{A}_2)\boldsymbol{x}$. Lee *et al.* showed that this approach can be generalized to an arbitrary number of servers.

Example 1 (Runtime distribution of coded and uncoded computing)**.** *In Fig. 2.4, we plot the probability density function of the runtime for coded and uncoded distributed matrix multiplication. We plot the distribution for a system composed of* 27 *servers. In the coded case, the input matrix is split into* 18 *submatrices that are encoded using a* $(27, 18)$ *MDS code, i.e., the final computation output can be decoded from the results returned by any* 18 *servers. In the uncoded case, the input matrix is split into* 27 *submatrices. Thus, each server has to do* $\frac{27}{18}$ *times more work in the* *coded case than in the uncoded case. The time axis is scaled by the complexity of the computation.*

The runtime distribution of the subtasks running on each server is a shifted exponential (see Section 2.3). Furthermore, coding increases the subtask runtime as each server is assigned a larger submatrix. However, the runtime of the first 18 *out of* 27 *servers to finish their respective subtasks in the coded case is a Gamma distributed random variable whose mean is lower than the average runtime of the uncoded computation, i.e.,*

$$f\left(\frac{27}{18}\sigma, 27, 18\right) < f(\sigma, 27, 27),$$

where f is defined in (2.1)*. Coding has thus sped up the computation. Furthermore, the variance of the runtime is significantly lower for the coded computation than for the uncoded. Note that we are not yet taking the decoding time into account.*

2.6 The Unified Scheme

Li *et al.* recently proposed a coded framework for the distributed matrix multiplication problem that unifies the ideas of [7, 8] in [9]. The framework is based on combining an erasure correcting code as suggested in [8] with repeated computations to enable coded multicasting as in [7]. The ideas of [7, 8] can be seen as particular instances of the framework in [9], corresponding to the minimization of the communication load or the computational delay.

In particular, [9] suggests the use of MDS codes, whose dimension is equal to the number of rows of $\boldsymbol{A}$, to generate some redundant computations. This is in contrast to the scheme of [8] where the code length is proportional to the number of servers. More precisely, for an $m \times n$ matrix $\boldsymbol{A}$, the authors of [9] proposed computing the coded matrix $\boldsymbol{C}$ by multiplying $\boldsymbol{A}$ with an (r, m) MDS encoding matrix $\boldsymbol{\Psi}_{\mathsf{MDS}}$, i.e., $\boldsymbol{C} = \boldsymbol{\Psi}_{\mathsf{MDS}}\boldsymbol{A}$. We denote the rows of the coded matrix $\boldsymbol{C}$ by $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_r$. Due to the MDS property, the source matrix $\boldsymbol{A}$ can be recovered from any m out of r unique coded rows $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_r$. Furthermore, for some input vector $\boldsymbol{x}$ of length n the output vector $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ can be recovered from any m out of r values $\boldsymbol{c}_1\boldsymbol{x}, \ldots, \boldsymbol{c}_r\boldsymbol{x}$.

In practice, the size of $\boldsymbol{A}$ can be very large. For example, Google performs matrix-vector multiplications with matrices of dimension of the order of $10^{10} \times 10^{10}$ when ranking the importance of websites [18]. Since the decoding complexity of MDS codes on the packet erasure channel is quadratic (for RS codes) in the code length [19], for very large matrix sizes the decoding complexity may be prohibitively high.

3
Coding for Distributed Computing

We consider the problem of multiplying a set of vectors with a matrix. In particular, given an $m \times n$ matrix $\boldsymbol{A} \in \mathbb{F}^{m\times n}$ and N vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{F}^n$, where $\mathbb{F}$ is some field, we want to compute the N vectors $\boldsymbol{y}_1 = \boldsymbol{A}\boldsymbol{x}_1, \ldots, \boldsymbol{y}_N = \boldsymbol{A}\boldsymbol{x}_N$. The computation is performed in a distributed fashion using K servers, $S_1, \ldots, S_K$. Each server stores μm matrix rows, for some $\frac{1}{K} \leq \mu \leq 1$. We refer to μ as the fraction of rows stored at each server and we assume that μ is selected such that μm is integer. Prior to distributing the rows among the servers, $\boldsymbol{A}$ is encoded by an $r \times m$ encoding matrix $\boldsymbol{\Psi}$, resulting in the coded matrix $\boldsymbol{C} = \boldsymbol{\Psi}\boldsymbol{A}$, of size $r \times n$, i.e., the rows of $\boldsymbol{A}$ are encoded using an (r, m) linear code with $r \geq m$. This encoding is used to alleviate the straggler problem as discussed in Section 2.5.

We allow assigning each row of the coded matrix $\boldsymbol{C}$ to several servers to enable coded multicasts (see Section 2.4). Let $q = K\frac{m}{r}$, where we assume that r divides Km and hence q is an integer. The r coded rows of $\boldsymbol{C}$, $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_r$, are divided into $\binom{K}{\mu q}$ disjoint batches, each containing $r/\binom{K}{\mu q}$ coded rows. Each batch is assigned to μq servers. Correspondingly, a batch B is labeled by a unique set $\mathcal{T} \subset \{S_1, \ldots, S_K\}$, of size $|\mathcal{T}| = \mu q$, denoting the subset of servers that store that batch. We write $B_{\mathcal{T}}$ to denote the batch stored at the unique set of servers $\mathcal{T}$. Server S_k, $k = 1, \ldots, K$, stores the coded rows of $B_{\mathcal{T}}$ if and only if $S_k \in \mathcal{T}$.

3.1 Distributed Computing Model

We consider the coded computing framework introduced in [9], which extends the MapReduce [6] framework described in Section 2.2. Specifically, the *map*, *shuffle*, and *reduce* phases are augmented to make use of the coded multicasting strategy proposed in [7] and the coded scheme proposed in [8] to alleviate the straggler problem. We assume that the input vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ are known to all servers at the start of the computation. The overall computation then proceeds in the following manner.

3.1.1 Map Phase

In the map phase, we compute in a distributed fashion coded intermediate values, which will be later used to obtain vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$. Server S multiplies the input vectors $\boldsymbol{x}_j$, $j = 1, \ldots, N$, by all the coded rows of matrix $\boldsymbol{C}$ it stores, i.e., it computes

$$\mathcal{Z}_j^{(S)} = \{\boldsymbol{c}\boldsymbol{x}_j :\ \forall \boldsymbol{c} \in \{B_{\mathcal{T}} :\ S \in \mathcal{T}\}\},\ j = 1, \ldots, N.$$

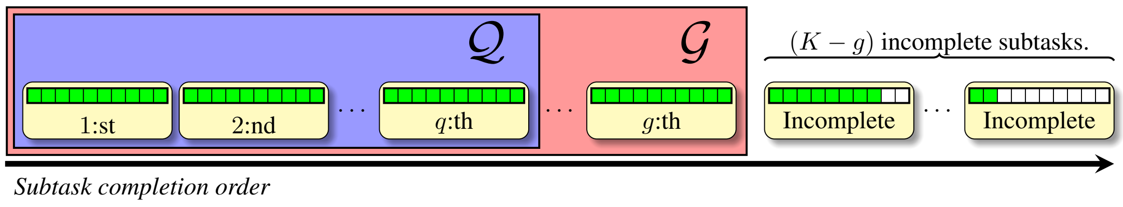


Figure 3.1: Servers (yellow boxes) finish their respective subtasks in random order.

The map phase terminates when a set of servers $\mathcal{G} \subseteq \{S_1, \ldots, S_K\}$ that collectively store enough values to decode the output vectors have finished their map computations. We denote the cardinality of $\mathcal{G}$ by g. The (r, m) linear code proposed in [9] is an MDS code for which $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$ can be obtained from any subset of q servers, i.e., $g = q$. We illustrate the completion of subtasks in Fig. 3.1.

We define the computational delay of the map phase as its average runtime per source row and vector $\boldsymbol{y}$, i.e.,

$$D_{\mathsf{map}} = \frac{1}{mN} f\left(\frac{\sigma_{\mathsf{map}}}{K}, K, g\right).$$

D_{map} is referred to simply as the computational delay in [9]. As all K servers compute μm inner products, each requiring n multiplications for each of the N input vectors, we have $\sigma_{\mathsf{map}} = K\mu mnN$.

After the map phase, the computation of $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$ proceeds using only the servers in $\mathcal{G}$. We denote by $\mathcal{Q} \subseteq \mathcal{G}$ the set of the first q servers to complete the map phase. Each of the q servers in $\mathcal{Q}$ is responsible to compute N/q of the vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$. Let $\mathcal{W}_S$ be the set containing the indices of the vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$ server $S \in \mathcal{Q}$ is responsible for. The remaining servers in $\mathcal{G}$ assist the servers in $\mathcal{Q}$ in the shuffle phase.

3.1.2 Shuffle Phase

In the shuffle phase, intermediate values calculated in the map phase are exchanged between servers in $\mathcal{G}$ until all servers in $\mathcal{Q}$ hold enough values to compute the vectors they are responsible for. As in [9], we allow creating and multicasting coded messages that are simultaneously useful for multiple servers. At a high level the shuffle phase proceeds in three steps:

1. Coded messages composed of several intermediate values are multicasted among the servers in $\mathcal{Q}$.

2. Intermediate values are unicasted among the servers in $\mathcal{Q}$.

3. Any intermediate values still missing from servers in $\mathcal{Q}$ are unicasted from the remaining servers in $\mathcal{G}$, i.e., from the servers in $\mathcal{G} \setminus \mathcal{Q}$.

For a subset of servers $\mathcal{S} \subset \mathcal{Q}$ and $S \in \mathcal{Q} \setminus \mathcal{S}$, we denote the set of intermediate values needed by server S and known *exclusively* by the servers in $\mathcal{S}$ by $\mathcal{V}_{\mathcal{S}}^{(S)}$. More formally,

$$\mathcal{V}_{\mathcal{S}}^{(S)} \triangleq \{\boldsymbol{c}\boldsymbol{x}_j : j \in \mathcal{W}_S \text{ and } \boldsymbol{c} \in \{B_{\mathcal{T}} : \mathcal{T} \cap \mathcal{Q} = \mathcal{S}\}\}.$$

We transmit coded multicasts only between the servers in $\mathcal{Q}$. Each coded message is simultaneously sent to j servers. Let $\alpha_j \triangleq \frac{\binom{q-1}{j}\binom{K-q}{\mu q-j}}{\frac{q}{K}\binom{K}{\mu q}}$. We denote by s_q the smallest number of recipients of a coded message,

$$s_q \triangleq \inf\left(s : \sum_{l=s}^{\mu q} \alpha_l \leq 1-\mu \right).$$

More specifically, for each $j \in \{\mu q, \mu q-1, \dots, s_q\}$, and every subset $\mathcal{S} \subseteq \mathcal{Q}$ of size $j+1$, the shuffle phase proceeds as follows.

1. For each $S \in \mathcal{S}$, we evenly and arbitrarily split $\mathcal{V}^{(S)}_{\mathcal{S}\setminus S}$ into j disjoint segments $\mathcal{V}^{(S)}_{\mathcal{S}\setminus S} = \{\mathcal{V}^{(S)}_{\mathcal{S}\setminus S,\tilde{S}} : \tilde{S} \in \mathcal{S}\setminus S\}$, and associate the segment $\mathcal{V}^{(S)}_{\mathcal{S}\setminus S,\tilde{S}}$ with server $\tilde{S} \in \mathcal{S}\setminus S$.
2. Server $\tilde{S} \in \mathcal{S}$ multicasts the bit-wise XOR of all the segments associated with it in $\mathcal{S}$. More precisely, it multicasts $\oplus_{S\in\mathcal{S}\setminus\tilde{S}}\, \mathcal{V}^{(S)}_{\mathcal{S}\setminus S,\tilde{S}}$ to the other servers in $\mathcal{S}\setminus\tilde{S}$.

Each recipient of a coded message has computed locally all values the message is composed of except for one. More precisely, for every pair of servers $S, \tilde{S} \in \mathcal{S}$, since server S has computed locally the segments $\mathcal{V}^{(S')}_{\mathcal{S}\setminus S',\tilde{S}}$ for all $S' \in \mathcal{S}\setminus\{\tilde{S}, S\}$, it can cancel them from the message sent by server $\tilde{S}$, and recover the intended segment. We finish the shuffle phase by either unicasting any remaining needed values until all servers in $\mathcal{Q}$ hold enough intermediate values to decode successfully, or by repeating the above two steps for $j = s_q - 1$, selecting the strategy achieving the lower communication load.

Definition 1. *The communication load, denoted by L, is the number of messages per source row and vector $\boldsymbol{y}$ exchanged during the shuffle phase, i.e., the total number of messages sent during the shuffle phase divided by mN.*

The communication load after completing the shuffle phase is given in [9]. If the shuffle phase finishes by unicasting the remaining needed values, the communication load after completing the multicast phase is

$$\sum_{j=s_q}^{\mu q} \frac{\alpha_j}{j}.$$

If instead steps 1 and 2 are repeated for $j = s_q - 1$, the communication load is

$$\sum_{j=s_q-1}^{\mu q} \frac{\alpha_j}{j}.$$

For the scheme in [9], the total communication load is

$$L_{\text{MDS}} = \min\left(\sum_{j=s_q}^{\mu q} \frac{\alpha_j}{j} + 1 - \mu - \sum_{j=s_q}^{\mu q} \alpha_j,\ \sum_{j=s_q-1}^{\mu q} \frac{\alpha_j}{j} \right). \tag{3.1}$$

As in [9], we consider the cost of a multicast message to be equal to that of a unicast message. In real systems, however, it may vary depending on the network architecture.

3.1.3 Reduce Phase

Finally, in the reduce phase, the vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ are computed. More specifically, server $S \in \mathcal{Q}$ uses the locally computed sets $\mathcal{Z}_1^{(S)}, \dots, \mathcal{Z}_N^{(S)}$ and the received messages to compute the vectors $\boldsymbol{y}_j$, $\forall j \in \mathcal{W}_S$. The computational delay of the reduce phase is its average runtime per source row and output vector $\boldsymbol{y}$, i.e.,

$$D_{\mathsf{reduce}} = \frac{1}{mN} f\left(\frac{\sigma_{\mathsf{reduce}}}{q}, q, q\right),$$

where σ_{reduce} is the computational complexity (see Section 2.3) of the reduce phase.

Definition 2. *The overall computational delay, D, is the sum of the map and reduce phase delays, i.e., $D = D_{\mathsf{map}} + D_{\mathsf{reduce}}$.*

3.2 Coded Computing Schemes

In this section, we formally define the coded computing schemes of [7–9] in terms of our system model. We refer to the scheme in [7] as the coded MapReduce (CMR) scheme, that in [8] as the straggler coding (SC) scheme, and the scheme in [9] as the unified scheme. We also define the uncoded computing (UC) scheme. The system model we consider consists of two separate coding theoretical components: An erasure correcting code used to alleviate the straggler problem and a structure of repeated computations used to create coded multicasting opportunities. To accurately assess the performance impact of each component we need to compare the performance of the combined scheme with that of the CMR and SC schemes, i.e., the corresponding coded scheme using only either component. Specifically, for a coded computing system with parameters K, q, m, and μ, we define the corresponding UC, CMR, SC, and unified schemes. These corresponding schemes are used for comparison purposes. In particular, we compare against the corresponding CMR scheme to analyze the impact of the erasure correcting code on the computational delay and with the corresponding SC scheme to understand the impact of the repeated computations on the load and the delay. When referring to system parameters of a corresponding UC, CMR, SC, or unified scheme, we write the scheme name in the subscript. We only explicitly mention the parameters that differ. The number of servers K is unchanged for all schemes considered.

3.2.1 Uncoded Computing

The UC scheme uses no erasure correcting coding and no coded multicasting. This is the traditional scheme used for distributed matrix multiplication. We measure the performance of other (coded) computing schemes primarily relative to the performance of the uncoded scheme. We define the corresponding uncoded scheme as the system with parameters $\mu_{\text{UC}} = \frac{1}{K}$ and $q_{\text{UC}} = K$, implying $\mu_{\text{UC}} q_{\text{UC}} = 1$. Furthermore, the encoding matrix $\boldsymbol{\Psi}_{\text{UC}}$ is the $m \times m$ identity matrix and the coded matrix is $\boldsymbol{C}_{\text{UC}} = \boldsymbol{A}$.

3.2.2 Coded MapReduce

The CMR scheme [7] uses only coded multicasting, i.e., $\boldsymbol{C}_{\mathrm{CMR}} = \boldsymbol{A}$ and $q_{\mathrm{CMR}} = K$. Furthermore, the fraction of rows stored at each server is $\mu_{\mathrm{CMR}} = \frac{\mu q}{K}$. We remark that there is no reduce delay for this scheme, i.e., $D_{\mathsf{reduce}} = 0$.

3.2.3 Straggler Coding

The SC scheme [8] uses an erasure correcting code but no coded multicasting. For the corresponding SC scheme, the code rate is unchanged, i.e., $q_{\mathrm{SC}} = q$ and the fraction of rows stored at each server is $\mu_{\mathrm{SC}} = \frac{1}{q_{\mathrm{SC}}}$. The encoding $\boldsymbol{\Psi}$ of the SC scheme is equivalent to splitting the rows of $\boldsymbol{A}$ into q_{SC} equally tall submatrices $\boldsymbol{A}_1, \dots, \boldsymbol{A}_{q_{\mathrm{SC}}}$ and applying a (K, q_{SC}) MDS code to the elements of each submatrix, thereby creating K coded submatrices $\boldsymbol{C}_1, \dots, \boldsymbol{C}_K$. The coded matrix $\boldsymbol{C}_{\mathrm{SC}}$ is the concatenation of $\boldsymbol{C}_1, \dots, \boldsymbol{C}_K$, i.e.,

$$\boldsymbol{C}_{\mathrm{SC}} = \begin{bmatrix} \boldsymbol{C}_1 \\ \vdots \\ \boldsymbol{C}_K \end{bmatrix}.$$

3.2.4 The Unified Scheme

The unified scheme [9] uses both erasure correcting codes and coded multicasting. The system parameters of the corresponding unified scheme are all unchanged. Furthermore, the encoding matrix, $\boldsymbol{\Psi}_{\text{unified}}$, of the unified scheme is an (r, m) MDS code encoding matrix.

4 Block-Diagonal Coding

We introduce a block-diagonal encoding matrix of the form

$$\boldsymbol{\Psi} = \begin{bmatrix} \boldsymbol{\psi}_1 & & \\ & \ddots & \\ & & \boldsymbol{\psi}_T \end{bmatrix},$$

where $\boldsymbol{\psi}_1, \dots, \boldsymbol{\psi}_T$ are $\frac{r}{T} \times \frac{m}{T}$ encoding matrices of an $(\frac{r}{T}, \frac{m}{T})$ MDS code, for some integer T that divides m and r. Note that the encoding given by $\boldsymbol{\Psi}$ amounts to partitioning the rows of $\boldsymbol{A}$ into T disjoint submatrices $\boldsymbol{A}_1, \dots, \boldsymbol{A}_T$ and encoding each submatrix separately. We refer to an encoding $\boldsymbol{\Psi}$ with T disjoint submatrices as a T-partitioned scheme, and to the submatrix of $\boldsymbol{C} = \boldsymbol{\Psi}\boldsymbol{A}$ corresponding to $\boldsymbol{\psi}_i$ as the i-th partition. We remark that all submatrices can be encoded using the same encoding matrix, i.e., $\boldsymbol{\psi}_i = \boldsymbol{\psi}$, $i = 1, \dots, T$, reducing the storage requirements, and encoding/decoding can be performed in parallel if many servers are available. We further remark that the case $\boldsymbol{\Psi} = \boldsymbol{\psi}$ (i.e., the number of partitions is $T = 1$) corresponds to the scheme in [9], which we will sometimes refer to as the *unpartitioned* scheme. We illustrate the block-diagonal encoding scheme with $T = 3$ partitions in Fig. 4.1.

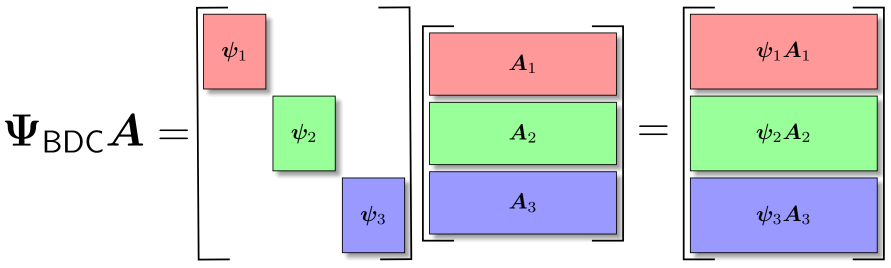

$\boldsymbol{\Psi}_{\mathsf{BDC}}\boldsymbol{A} =$
$\boldsymbol{\psi}_1$
$\boldsymbol{\psi}_2$
$\boldsymbol{\psi}_3$
$\boldsymbol{A}_1$
$\boldsymbol{A}_2$
$\boldsymbol{A}_3$
$=$
$\boldsymbol{\psi}_1\boldsymbol{A}_1$
$\boldsymbol{\psi}_2\boldsymbol{A}_2$
$\boldsymbol{\psi}_3\boldsymbol{A}_3$

Figure 4.1: BDC scheme for $T = 3$ partitions.

4.1 Assignment of Coded Rows to Batches

For a block-diagonal encoding matrix $\boldsymbol{\Psi}$, we denote by $\boldsymbol{c}_i^{(t)}$, $t = 1, \dots, T$ and $i = 1, \dots, r/T$, the i-th coded row of $\boldsymbol{C}$ within partition t. For example, $\boldsymbol{c}_1^{(2)}$ denotes the first coded row of the second partition. As described in Chapter 3, the coded rows are divided into $\binom{K}{\mu q}$ disjoint batches. To formally describe the assignment of coded rows to batches we use a $\binom{K}{\mu q} \times T$ integer matrix $\boldsymbol{P} = [p_{i,j}]$, where $p_{i,j}$

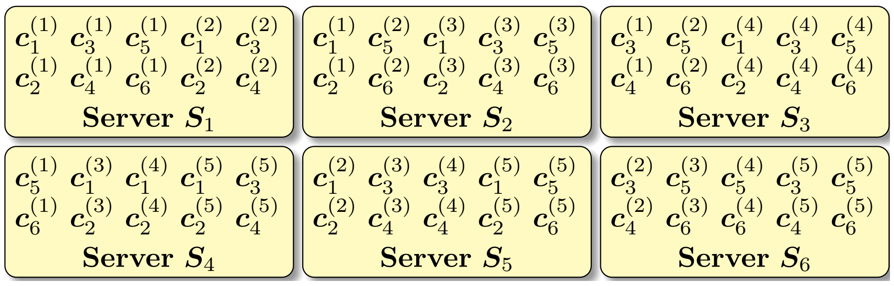

$c_1^{(1)}$ $c_3^{(1)}$ $c_5^{(1)}$ $c_1^{(2)}$ $c_3^{(2)}$
$c_2^{(1)}$ $c_4^{(1)}$ $c_6^{(1)}$ $c_2^{(2)}$ $c_4^{(2)}$
Server S_1
$c_1^{(1)}$ $c_5^{(2)}$ $c_1^{(3)}$ $c_3^{(3)}$ $c_5^{(3)}$
$c_2^{(1)}$ $c_6^{(2)}$ $c_2^{(3)}$ $c_4^{(3)}$ $c_6^{(3)}$
Server S_2
$c_3^{(1)}$ $c_5^{(2)}$ $c_1^{(4)}$ $c_3^{(4)}$ $c_5^{(4)}$
$c_4^{(1)}$ $c_6^{(2)}$ $c_2^{(4)}$ $c_4^{(4)}$ $c_6^{(4)}$
Server S_3
$c_5^{(1)}$ $c_1^{(3)}$ $c_1^{(4)}$ $c_1^{(5)}$ $c_3^{(5)}$
$c_6^{(1)}$ $c_2^{(3)}$ $c_2^{(4)}$ $c_2^{(5)}$ $c_4^{(5)}$
Server S_4
$c_1^{(2)}$ $c_3^{(3)}$ $c_3^{(4)}$ $c_1^{(5)}$ $c_5^{(5)}$
$c_2^{(2)}$ $c_4^{(3)}$ $c_4^{(4)}$ $c_2^{(5)}$ $c_6^{(5)}$
Server S_5
$c_3^{(2)}$ $c_5^{(3)}$ $c_5^{(4)}$ $c_3^{(5)}$ $c_5^{(5)}$
$c_4^{(2)}$ $c_6^{(3)}$ $c_6^{(4)}$ $c_4^{(5)}$ $c_6^{(5)}$
Server S_6

Figure 4.2: Storage design for $m = 20$, $N = 4$, $K = 6$, $q = 4$, $\mu = 1/2$, and $T = 5$.

describes the number of rows from partition j that are stored in batch i. Note that, due to the MDS property, any set of m/T rows of a partition is sufficient to decode the partition. Thus, without loss of generality, we consider a *sequential* assignment of rows of a partition into batches. More precisely, when first assigning a row of partition t to a batch, we pick $\boldsymbol{c}_1^{(t)}$. Next time a row of partition t is assigned to a batch we pick $\boldsymbol{c}_2^{(t)}$, and so on. In this manner, each coded row is assigned to a unique batch exactly once. For example, for the assignment $\boldsymbol{P}$ in Example 2 (see (4.1)), rows $\boldsymbol{c}_1^{(1)}$ and $\boldsymbol{c}_2^{(1)}$ are assigned to batch 1, $\boldsymbol{c}_3^{(1)}$ and $\boldsymbol{c}_4^{(1)}$ are assigned to batch 2, and so on. The rows of $\boldsymbol{P}$ are labeled by the subset of servers the corresponding batch is stored at, and the columns are labeled by its partition index. We refer to the pair $(\boldsymbol{\Psi}, \boldsymbol{P})$ as the *storage design*. The assignment matrix $\boldsymbol{P}$ must satisfy the following conditions.

1. The entries of each row of $\boldsymbol{P}$ must sum to the batch size, i.e.,

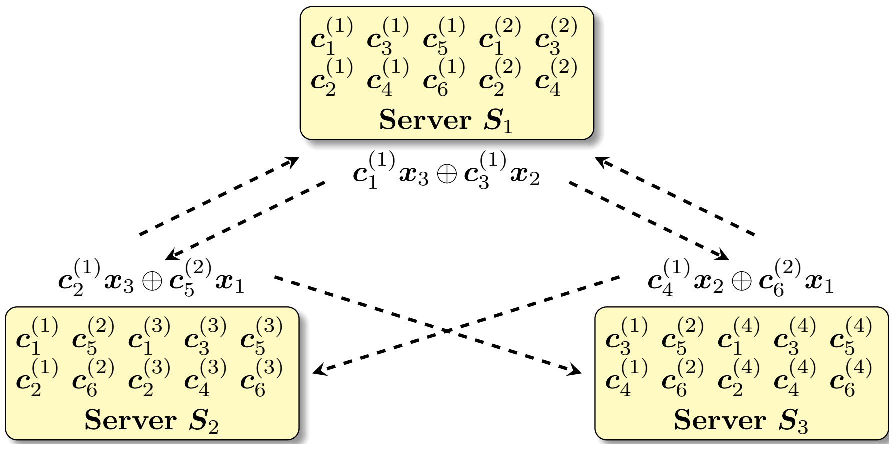
$$\sum_{j=1}^{T} p_{i,j} = r/\binom{K}{\mu q},\ 1 \leq i \leq \binom{K}{\mu q}.$$

2. The entries of each column of $\boldsymbol{P}$ must sum to the number of rows per partition, i.e.,

$$\sum_{i=1}^{\binom{K}{\mu q}} p_{i,j} = \frac{r}{T},\ 1 \leq j \leq T.$$

Example 2 ($m = 20$, $N = 4$, $K = 6$, $q = 4$, $\mu = 1/2$, $T = 5$)**.** *For these parameters, there are $r/T = 6$ coded rows per partition, of which $m/T = 4$ are sufficient for decoding, and $\binom{K}{\mu q} = 15$ batches, each containing $r/\binom{K}{\mu q} = 2$ coded rows. We construct the storage design shown in Fig. 4.2 with assignment matrix*

$$\boldsymbol{P} = \begin{array}{c} \\ (S_1,S_2) \\ (S_1,S_3) \\ (S_1,S_4) \\ (S_1,S_5) \\ \vdots \\ (S_4,S_6) \\ (S_5,S_6) \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ & & \vdots & & \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix} \end{array}, \tag{4.1}$$



$\boldsymbol{c}_1^{(1)}$ $\boldsymbol{c}_3^{(1)}$ $\boldsymbol{c}_5^{(1)}$ $\boldsymbol{c}_1^{(2)}$ $\boldsymbol{c}_3^{(2)}$
$\boldsymbol{c}_2^{(1)}$ $\boldsymbol{c}_4^{(1)}$ $\boldsymbol{c}_6^{(1)}$ $\boldsymbol{c}_2^{(2)}$ $\boldsymbol{c}_4^{(2)}$
Server S_1
$\boldsymbol{c}_1^{(1)}\boldsymbol{x}_3 \oplus \boldsymbol{c}_3^{(1)}\boldsymbol{x}_2$
$\boldsymbol{c}_2^{(1)}\boldsymbol{x}_3 \oplus \boldsymbol{c}_5^{(2)}\boldsymbol{x}_1$
$\boldsymbol{c}_4^{(1)}\boldsymbol{x}_2 \oplus \boldsymbol{c}_6^{(2)}\boldsymbol{x}_1$
$\boldsymbol{c}_1^{(1)}$ $\boldsymbol{c}_5^{(2)}$ $\boldsymbol{c}_1^{(3)}$ $\boldsymbol{c}_3^{(3)}$ $\boldsymbol{c}_5^{(3)}$
$\boldsymbol{c}_2^{(1)}$ $\boldsymbol{c}_6^{(2)}$ $\boldsymbol{c}_2^{(3)}$ $\boldsymbol{c}_4^{(3)}$ $\boldsymbol{c}_6^{(3)}$
Server S_2
$\boldsymbol{c}_3^{(1)}$ $\boldsymbol{c}_5^{(2)}$ $\boldsymbol{c}_1^{(4)}$ $\boldsymbol{c}_3^{(4)}$ $\boldsymbol{c}_5^{(4)}$
$\boldsymbol{c}_4^{(1)}$ $\boldsymbol{c}_6^{(2)}$ $\boldsymbol{c}_2^{(4)}$ $\boldsymbol{c}_4^{(4)}$ $\boldsymbol{c}_6^{(4)}$
Server S_3

Figure 4.3: Multicasting coded values between servers S_1, S_2, and S_3.

where rows are labeled by the subset of servers the batch is stored at, and columns are labeled by the partition index. For this storage design, any $g = 4$ *servers collectively store at least 4 coded rows from each partition. However, some servers store more rows than needed to decode some partitions, suggesting that this storage design is suboptimal.*

Assume $\mathcal{G} = \{S_1, S_2, S_3, S_4\}$ *is the set of* $g = 4$ *servers that finish their map computations first. Also, assign vector* $\boldsymbol{y}_i$ *to server* S_i, $i = 1, 2, 3, 4$. *We illustrate the coded shuffling scheme for* $\mathcal{S} = \{S_1, S_2, S_3\}$ *in Fig. 4.3.* S_1 *multicasts* $\boldsymbol{c}_1^{(1)}\boldsymbol{x}_3 \oplus \boldsymbol{c}_3^{(1)}\boldsymbol{x}_2$ *to* S_2 *and* S_3. *Since* S_2 *and* S_3 *can cancel* $\boldsymbol{c}_1^{(1)}\boldsymbol{x}_3$ *and* $\boldsymbol{c}_3^{(1)}\boldsymbol{x}_2$, *respectively, both servers receive one needed intermediate value. Similarly,* S_2 *multicasts* $\boldsymbol{c}_2^{(1)}\boldsymbol{x}_3 \oplus \boldsymbol{c}_5^{(2)}\boldsymbol{x}_1$, *while* S_3 *multicasts* $\boldsymbol{c}_4^{(1)}\boldsymbol{x}_2 \oplus \boldsymbol{c}_6^{(2)}\boldsymbol{x}_1$. *This process is repeated for* $\mathcal{S} = \{S_2, S_3, S_4\}$, $\mathcal{S} = \{S_1, S_3, S_4\}$, *and* $\mathcal{S} = \{S_1, S_2, S_4\}$. *After the shuffle phase, we have sent* 12 *multicast messages and* 30 *unicast messages, resulting in a communication load of* $(12 + 30)/20/4 = 0.525$, *a* 50% *increase from the load of the unpartitioned scheme (*0.35, *given by* (3.1)*). In this case,* S_1 *received additional intermediate values from partition* 2, *despite already storing enough, further indicating that the assignment in* (4.1) *is suboptimal.*

4.2 Performance of the Block-Diagonal Coding

In this section, we analyze the impact of partitioning on the performance. In the following theorem, we show that we can partition up to the batch size without increasing the communication load and the computational delay of the map phase with respect to the original scheme in [9].

Theorem 1. *For* $T \leq r/\binom{K}{\mu q}$ *there exists an assignment matrix* $\boldsymbol{P}$ *such that the communication load and computational delay of the map phase are equal to those of the unpartitioned scheme.*

Proof. The computational delay of the map phase is equal to that of the unpartitioned scheme if any q servers hold enough coded rows to decode all partitions. For $T = r/\binom{K}{\mu q}$ we let $\boldsymbol{P}$ be a $\binom{K}{\mu q} \times T$ all-ones matrix and show that it has this property by repeating the argument from [9, Sec. IV.B] for each partition. In this case, any set of q servers collectively store $\frac{\mu q m}{T}$ rows from each partition, and since each coded row is stored by at most μq servers, any q servers collectively store at least $\frac{\mu q m}{\mu q T} = \frac{m}{T}$ unique coded rows from each partition.

To see that the communication load is unchanged, we note that after having sent all multicast messages, all servers hold the same total number of coded intermediate values regardless of the level of partitioning. Since all servers need m/T values from each partition, if we can show that each server will hold the same number of coded values from each partition at this point, we have also shown that the communication load is unchanged. First, for the given assignment, all servers store the same number of rows from each partition before the shuffle phase. Second, in the shuffle phase, all servers $S \in \mathcal{Q}$ receive the values in $\mathcal{V}^{(S)}_{\mathcal{S}\setminus S}$ for all $\mathcal{S} \subseteq \mathcal{Q}$ such that $|\mathcal{S}| = \mu q + 1, \mu q, \dots, s_q + 1$. $\mathcal{V}^{(S)}_{\mathcal{S}\setminus S}$ is computed from a union of batches and therefore guaranteed to contain an equal number of coded intermediate values from each partition for this assignment. These arguments together show that all servers will hold an equal number of coded values from each partition after all multicast messages have been sent. For $T < r/\binom{K}{\mu q}$ we can split partitions into smaller partitions until we have exactly $r/\binom{K}{\mu q}$ partitions, and repeat the same argument. $\square$

4.2.1 Communication Load

For the unpartitioned scheme of [9], $\mathcal{G} = \mathcal{Q}$, and the number of remaining values that need to be unicasted after the multicast phase is constant, regardless which subset $\mathcal{Q}$ of servers finish first their map computations. However, for the block-diagonal (partitioned) coding scheme, both g and the number of remaining unicasts may vary.

For a given assignment $\boldsymbol{P}$ and a specific $\mathcal{Q}$, we denote by $U^{(S)}_{\mathcal{Q}}(\boldsymbol{P})$ the number of remaining values needed after the multicast phase by server $S \in \mathcal{Q}$, and by $U_{\mathcal{Q}}(\boldsymbol{P}) \triangleq \sum_{S\in\mathcal{Q}} U^{(S)}_{\mathcal{Q}}(\boldsymbol{P})$ the total number of remaining values needed by the servers in $\mathcal{Q}$. We remark that all sets $\mathcal{Q}$ are equally likely. Let $\mathbb{Q}^q$ denote the superset of all sets $\mathcal{Q}$. Furthermore, we denote by $L_{\mathbb{Q}}$ the average communication load of the messages that are unicasted after the multicasting step (see Section 3.1.2),

$$L_{\mathbb{Q}}(\boldsymbol{P}) \triangleq \frac{1}{mN}\frac{1}{|\mathbb{Q}^q|}\sum_{\mathcal{Q}\in\mathbb{Q}^q} U_{\mathcal{Q}}(\boldsymbol{P}).$$

Then, for a given storage design $(\boldsymbol{\Psi}, \boldsymbol{P})$, the communication load of the block-diagonal coding scheme is given by

$$L_{\mathrm{BDC}}(\boldsymbol{\Psi}, \boldsymbol{P}) = \min\left(\sum_{j=s_q}^{\mu q}\frac{\alpha_j}{j} + L_{\mathbb{Q}}(\boldsymbol{P}), \sum_{j=s_q-1}^{\mu q}\frac{\alpha_j}{j} + L_{\mathbb{Q}}(\boldsymbol{P})\right), \tag{4.2}$$

where $L_{\mathbb{Q}}(\boldsymbol{P})$ depends on the shuffling scheme (see Section 3.1.2) and is different in the first and second term of the minimization in (4.2). To evaluate $U^{(S)}_{\mathcal{Q}}$, we count the total number of intermediate values that need to be unicasted to server S until it holds m/T intermediate values from each partition.

For a given $\boldsymbol{\Psi}$, the assignment of rows into batches can be formulated as an optimization problem, where one would like to minimize L_{BDC} over all assignments $\boldsymbol{P}$. More precisely, the optimization problem is

$$\min_{\boldsymbol{P}\in\mathbb{P}} L_{\text{BDC}}(\boldsymbol{\Psi}, \boldsymbol{P}),$$

where $\mathbb{P}$ is the set of all assignments $\boldsymbol{P}$, and where the dependence of L_{BDC} on $\boldsymbol{P}$ is nonlinear. This is a computationally complex problem since both the complexity of evaluating the performance of a given assignment and the number of assignments scale exponentially in the problem size. We address this problem in Section 4.3.

4.2.2 Computational Delay

We consider the delay incurred by both the map and reduce phases (see Definition 2). We do not consider the delay incurred by the shuffle phase as the computations it requires are simple in comparison. Note that in [9] only D_{map} is considered, i.e., $D = D_{\mathsf{map}}$. However, one should not neglect the computational delay incurred by the reduce phase. Thus, one should consider the overall computational delay

$$D = D_{\mathsf{map}} + D_{\mathsf{reduce}}.$$

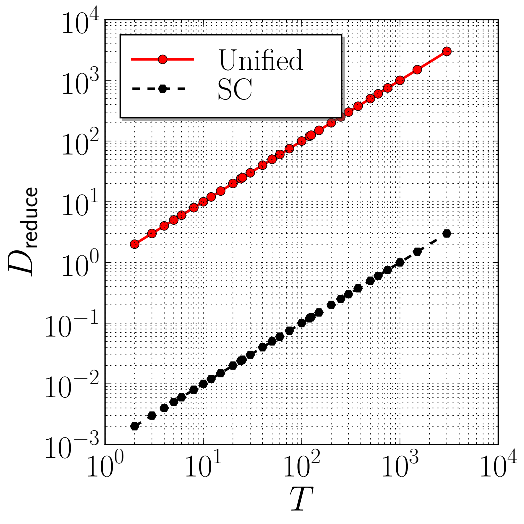
The reduce phase consists of decoding the N output vectors and hence the delay it incurs depends on the underlying code and decoding algorithm. We assume that each partition is encoded using an RS code and is decoded using the Berlekamp-Massey algorithm. We measure the decoding complexity by its associated shifted-exponential parameter σ (see Section 2.3).

The number of field multiplications required to decode an $(r/T, m/T)$ RS code is $(r/T)^2\epsilon$ [12], where ϵ is the fraction of erased symbols. With ϵ upperbounded by $1 - \frac{q}{K}$ (the map phase terminates when a fraction of at least $\frac{q}{K}$ symbols from each partition is available) the complexity of decoding the T partitions for all N output vectors is upperbounded as

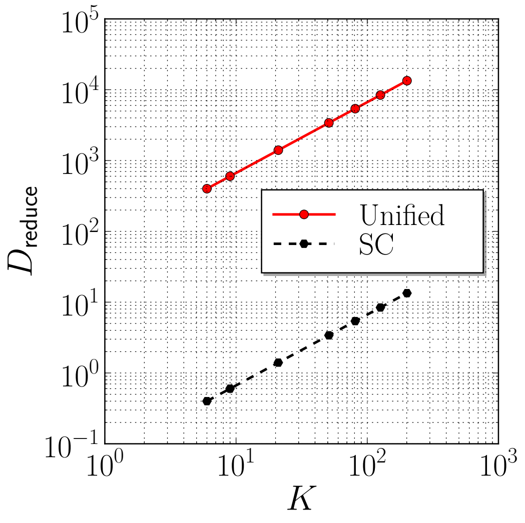
$$\sigma_{\mathsf{reduce}} \leq \frac{r^2(1-\frac{q}{K})N}{T}. \tag{4.3}$$

The decoding complexity of the unified scheme in [9] is given by evaluating (4.3) for $T = 1$. By choosing T close to r, we can thus significantly lower the delay of the reduce phase. On the other hand, the scheme in [8] uses codes of length proportional to the number of servers K. The decoding complexity of the SC scheme in [8] is thus given by evaluating (4.3) for $T = \frac{m}{q}$.

Example 3 (Reduce (decoding) delay). *In Fig. 4.4, we plot the reduce delay of the corresponding unified and SC schemes. We normalize the delay by that of our proposed scheme. In 4.4(a), we plot the normalized reduce delay as a function of the number of partitions T. The system parameters are $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. The reduce delay of our proposed scheme is between a factor 2 and 3000 lower than that of the unified scheme. On the other hand, the reduce delay of the SC scheme is lower than that of ours for $T \leq 1000$. For*



(a)



(b)

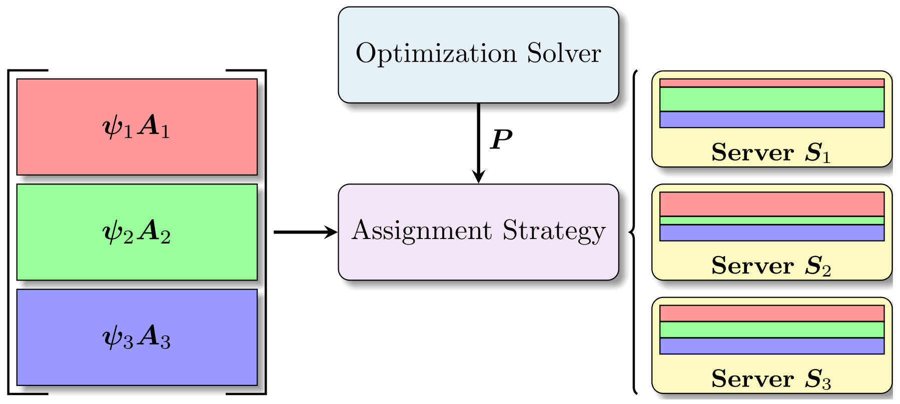
Figure 4.4: Reduce delay of the unified and SC schemes; normalized by that of our proposed scheme. **(a)** As a function of partitioning for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. **(b)** As a function of system size for $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$.

$T = 3000$*, the reduce delay of the SC scheme is about* 3 *times that of our proposed scheme.*

In 4.4(b), we plot the normalized reduce delay for a constant $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ *rows per partition, and code rate* $m/r = 2/3$ *as a function of the number of servers,* K*. The reduce delay of our proposed scheme is significantly lower than that of the unified scheme for all systems considered. For small systems (less than* $K = 15$ *servers) the reduce delay of the SC scheme is lower than that of ours. However, for the largest system considered (*$K = 201$ *servers) the reduce delay of the SC scheme is an order of magnitude higher than that of our proposed scheme.*

4.3 Assignment Solvers

We propose two solvers for the problem of assigning rows into batches: a heuristic solver that is fast even for large problem instances, and a hybrid solver combining the heuristic solver with a branch-and-bound solver. The branch-and-bound solver produces an optimal assignment but is significantly slower, hence it can be used as stand-alone only for small problem instances. We use a dynamic programming approach to speed up the branch-and-bound solver by caching $U_{\mathcal{Q}}^{(S)}$ for all $\mathcal{Q} \in \mathbb{Q}^q$, indexed by the batches each $U_{\mathcal{Q}}^{(S)}$ is computed from. This way we only need to update the affected $U_{\mathcal{Q}}^{(S)}$ when assigning a row to a batch. For all solvers, we first label the batches lexiographically and then optimize L_{BDC} in (4.2). We illustrate how the optimization solver fits into the computing scheme in Fig. 4.5. The solvers are available under the Apache 2.0 license [20].



$\psi_1 \boldsymbol{A}_1$
$\psi_2 \boldsymbol{A}_2$
$\psi_3 \boldsymbol{A}_3$
Optimization Solver
$\boldsymbol{P}$
Assignment Strategy
Server $\boldsymbol{S}_1$
Server $\boldsymbol{S}_2$
Server $\boldsymbol{S}_3$

Figure 4.5: Assignment of coded rows to servers.

4.3.1 Heuristic Solver

The heuristic solver is inspired by the assignment matrices created by the branch-and-bound solver for small instances. It creates an assignment matrix $\boldsymbol{P}$ in two steps. We first set each entry of $\boldsymbol{P}$ to $\gamma \triangleq \left\lfloor r / \left(\binom{K}{\mu q} \cdot T \right) \right\rfloor$, thus assigning the first $\binom{K}{\mu q}\gamma$ rows of each partition to batches such that each batch is assigned γT rows. Let $d = r/\binom{K}{\mu q} - \gamma T$ be the number of rows that still need to be assigned to each batch. The $r/T - \binom{K}{\mu q}\gamma$ rows per partition not assigned yet are assigned in the second step as given in Algorithm 1.

```
Algorithm 1: Heuristic Remaining Assignment
  Input   : $\boldsymbol{P}$, $d$, $K$, $T$, and $\mu q$
  for $0 \le a < d\binom{K}{\mu q}$ do
      $i \leftarrow \lfloor a/d \rfloor + 1$
      $j \leftarrow (a \bmod T) + 1$
      $p_{i,j} \leftarrow p_{i,j} + 1$
  end
  return $\boldsymbol{P}$
```

Example 4 (Heuristic solver)**.** *For the system in Example 2 with parameters $m = 20$, $N = 4$, $K = 6$, $q = 4$, $\mu = 1/2$, and $T = 5$, we have $\gamma = 0$, and the heuristic solver creates the assignment matrix*

$$\boldsymbol{P} = \begin{array}{c} \\ (S_1,S_2) \\ (S_1,S_3) \\ (S_1,S_4) \\ (S_1,S_5) \\ \vdots \\ (S_4,S_6) \\ (S_5,S_6) \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ & & \vdots & & \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} .$$

4.3.2 Branch-and-Bound Solver

The branch-and-bound solver finds an optimal solution by recursively branching at each batch for which there is more than one possible assignment and considering all options. The solver is initially given an empty assignment matrix, i.e., an all-zeros $\binom{K}{\mu q} \times T$ matrix. For each branch, we lowerbound the value of the objective function of any assignment in that branch and only investigate branches with possibly better assignments. The branch-and-bound operations given below are repeated until there are no more potentially better solutions to consider.

4.3.2.1 Branch

For the first row of $\boldsymbol{P}$ with remaining assignments, branch on every available assignment for that row. More precisely, find the smallest index i of a row of the assignment matrix $\boldsymbol{P}$ that does not sum to the batch size, i.e.,

$$\sum_{j=1}^{T} p_{i,j} < r / \binom{K}{\mu q}.$$

For row i, branch on incrementing the element $p_{i,j}$ by 1 for all column indices j that do not sum to the number of partitions, i.e.,

$$\sum_{i=1}^{\binom{K}{\mu q}} p_{i,j} < \frac{r}{T}.$$

4.3.2.2 Bound

We keep a record of all nonzero $U_{\mathcal{Q}}^{(S)}$ for all $\mathcal{Q}$ and S, and index them by the batches they are computed from. An assignment to a batch can at most reduce L_{BDC} by $1/\left(mN\left|\mathbb{Q}^q\right|\right)$ for each nonzero $U_{\mathcal{Q}}^{(S)}$ indexed by that batch, and we lowerbound L_{BDC} for a subtree by assuming that no $U_{\mathcal{Q}}^{(S)}$ will drop to zero for any subsequent assignment.

4.3.3 Hybrid Solver

The branch-and-bound solver can only be used on its own only for small instances. However, it can be used to complete a *partial* assignment matrix, i.e., a matrix $\boldsymbol{P}$ for which not all rows sum to the batch size. The branch-and-bound solver then completes the assignment optimally. We first find a candidate solution using the heuristic solver and then iteratively improve it using the branch-and-bound solver. In particular, we decrement by 1 a random set of elements of $\boldsymbol{P}$ and then use the branch-and-bound solver to reassign the corresponding rows optimally. We repeat this process until the average improvement between iterations drops below some threshold.

5 Numerical Results

In this chapter, we present numerical results for the schemes presented in [7–9] alongside those for our proposed BDC scheme. We also compare the performance of the systems with assignment $\boldsymbol{P}$ produced by the heuristic and hybrid solvers. All results are normalized by the performance of the corresponding uncoded scheme discussed in Section 3.2.1.

5.1 Coded Computing Comparison

In Fig. 5.1, we plot the performance of the CMR, SC, and unified schemes normalized by the performance of the uncoded scheme. We also give the performance of the proposed BDC scheme with assignment $\boldsymbol{P}$ given by the heuristic solver.

In 5.1(a), we plot the normalized communication load L (see Definition 1) and overall computational delay D (see Definition 2) as a function of the number of partitions T. The system parameters are $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. The parameters of the CMR scheme are $q_{\text{CMR}} = 9$ and $\mu_{\text{CMR}} = \frac{2}{9}$. The fraction of rows stored at each server for the SC scheme is $\mu_{\text{SC}} = \frac{1}{6}$. For up to $r/\binom{K}{\mu q} = 250$ partitions, the proposed scheme does not incur any loss in D_{map} and communication load with respect to the unified scheme (see Theorem 1). However, the proposed scheme yields a significantly lower overall computational delay compared to the unified scheme (about 40% speedup for $T > 50$). For heavy partitioning (around $T = 800$) a tradeoff between partitioning level, communication load, and map phase delay is observed. With 3000 partitions (the maximum possible), there is about a 10% increase in communication load. Note that the gain in overall computational delay saturates with the partitioning level, thus there is no reason to partition beyond a given level. The delay of the SC scheme is close to half that of our proposed scheme. However, it does not include redundant computations and thus has a communication load almost twice that of our scheme. Furthermore, the delay of our proposed scheme is close to 25% lower than that of the CMR scheme for $T > 50$ partitions.

In 5.1(b), we plot the normalized performance for a constant $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$ as a function of the number of servers, K. The results shown are averages over 1000 randomly generated realizations of $\mathcal{G}$ as it is computationally unfeasible to evaluate the performance exhaustively in this case. The communication load of our proposed scheme is within a few percentage points of both the CMR and unified schemes. The delay is lower than that of the CMR scheme, indicating that the erasure correcting

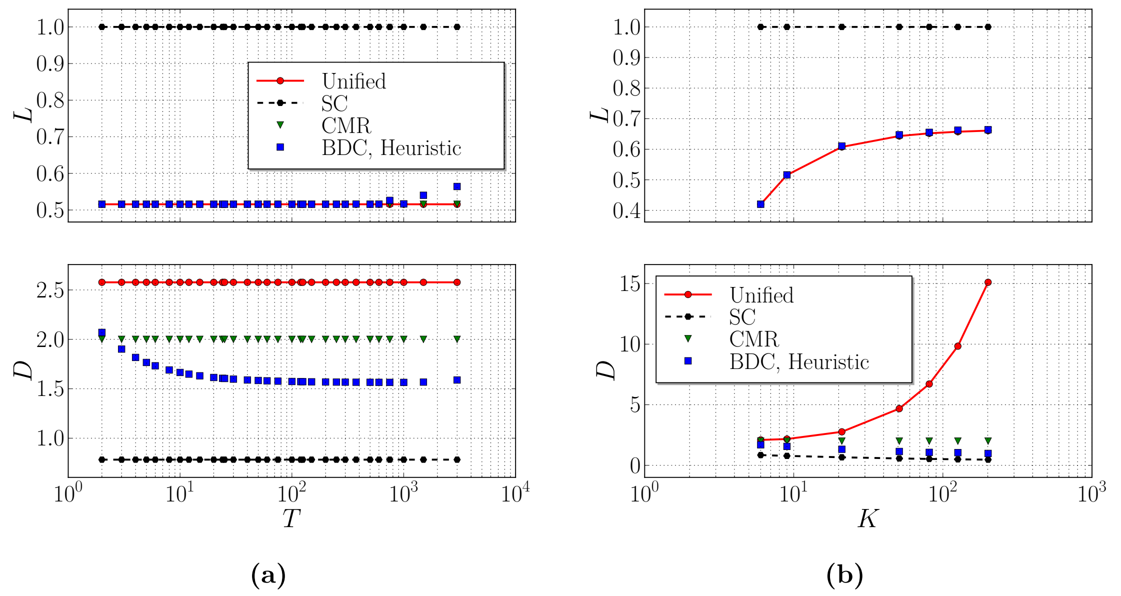


Figure 5.1: **(a)** The tradeoff between partitioning and performance for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. **(b)** Performance dependence on system size for $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$.

code provides a speedup. The overall computational delay of our proposed scheme is twice that of the SC scheme as each coded row is assigned to $\mu q = 2$ servers to create coded multicasting opportunities. On the other hand, the communication load of our proposed scheme is 42% that of the SC scheme for $K = 6$ and 66% that of the SC scheme for $K = 201$. Our proposed scheme outperforms the unified scheme for all sizes considered. For the largest system considered, the overall computational delay of the unified scheme is almost 15 times that of our proposed scheme. The delay of the unified scheme deteriorates rapidly with system size due to its high decoding complexity. In fact, most of the delay of the unified scheme is due to the decoding, even for relatively small systems.

5.2 Assignment Solver Comparison

In Fig. 5.2, we plot the performance of the proposed BDC scheme with assignment $\boldsymbol{P}$ given by the heuristic and the hybrid solvers. We normalize the performance by that of the uncoded scheme. We also give the average performance over 100 random assignments.

In 5.2(a), we plot the normalized communication load L and overall computational delay D as a function of the number of partitions T. The system parameters are $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. For T less than about 200, the performance of all solvers is identical. On the other hand, both the computational delay and the communication load are reduced with $\boldsymbol{P}$ from the heuristic solver over the random assignments for T larger than 200 (about 3 percentage points

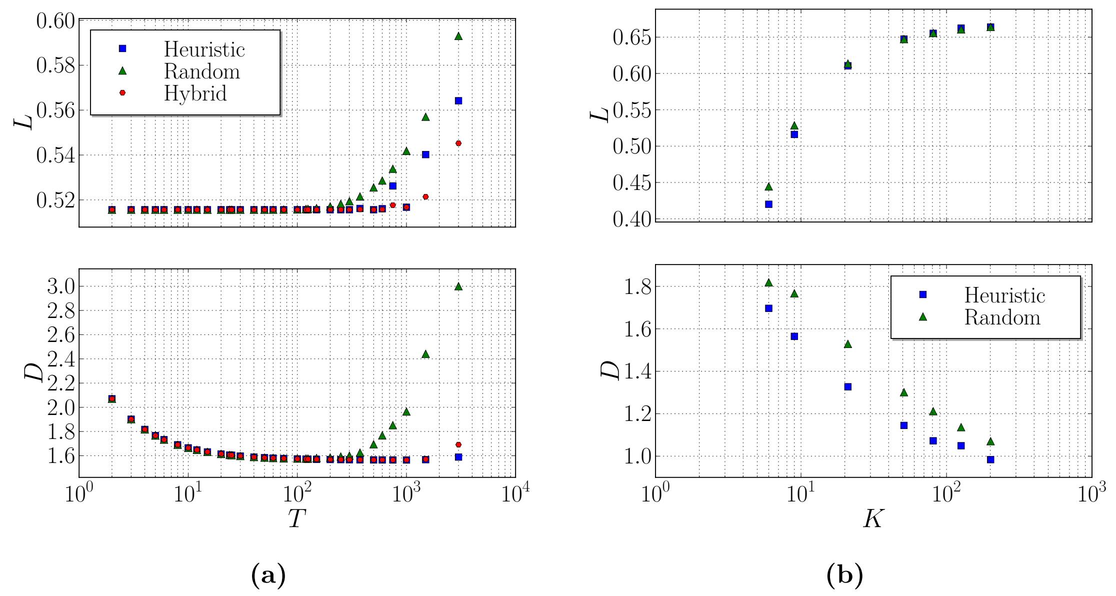


Figure 5.2: **(a)** Solver performance as a function of partitioning for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6$, and $\mu = 1/3$. **(b)** Performance dependence on system size for $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$.

for load and 140 percentage points for delay at $T = 3000$). A further improvement in communication load can be achieved using the hybrid solver, but at the expense of a possibly larger computational delay.

In 5.2(b), we plot the normalized performance for a constant $\mu q = 2$, $n = 10000$, $\mu m = 2000$, $m/T = 10$ rows per partition, and code rate $m/r = 2/3$ as a function of the number of servers, K. The results shown are averages over 1000 randomly generated realizations of $\mathcal{G}$ as it is computationally unfeasible to evaluate the performance exhaustively in this case. The heuristic solver outperforms the random assignments both in terms of load and delay for all system sizes considered. The difference is larger for small K and the performance difference is diminishing with larger K. The heuristic solver outperforms the random assignments by about 10 percentage points in terms of delay for the largest system considered. The hybrid solver is too computationally complex for use with the largest systems considered.

6
Conclusions and Future Work

In this thesis, we have explored codes for use in distributed computing systems. In particular, we have investigated the ability of the coded schemes of [7–9] to deal with the bandwidth scarcity and straggler problem in the context of distributed matrix multiplication. We see that the schemes in [7] and [8] are effective means of tackling the bandwidth scarcity and straggler problem, respectively. For instance, for a system with 9 servers and a 6000×6000 matrix the communication load using the scheme in [7] is close to half that of the uncoded computation. Alternatively, by instead using the scheme in [8] the overall computational delay (including the decoding latency) is about 50% that of the uncoded computation. The scheme in [9] provides a unified framework that combines the functionality of both schemes. However, we show that even for relatively small systems the decoding complexity of the MDS code proposed in [9] is prohibitively high. For the same system the overall computational delay of the unified scheme is more than 2.5x that of the uncoded computation. Furthermore, the difference is increasing with system size.

We have introduced a BDC scheme for distributed matrix multiplication based on partitioning the matrix into smaller submatrices and encoding each submatrix separately. Compared to the earlier scheme of [9], the proposed scheme yields a significantly lower overall computational delay with no increase in communication load up to a level of partitioning. For instance, for a square matrix with 6000 rows and columns, the proposed scheme reduces the computational delay by about 40% when the number of partitions $T > 50$. Essentially, our proposed scheme retains both the flexibility of the scheme in [9] and the performance of the schemes in [7,8].

We believe that there are many interesting coding schemes and tradeoffs still unexplored in this area. For example, we are currently considering rateless codes (e.g., Luby Transform codes [21] and Raptor codes [22]) as an alternative coding scheme with low decoding complexity. We would also like to investigate multi-stage computations, where the output of one coded computation is the input of another. Another interesting area is codes tailored to specific nonlinear algorithms. Furthermore, we would like to see more implementations of the coded schemes discussed herein to more accurately assess their performance impact for problems faced by industry.

Bibliography

[1] H. Sutter and J. Larus, “Software and the concurrency revolution,” *ACM Queue*, vol. 3/7, pp. 54–62, September 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/software-and-the-concurrency-revolution/

[2] M. Ben-Ari, *Principles of Concurrent and Distributed Programming.* Addison-Wesley, 2006.

[3] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool Publishers, 2009.

[4] C. L. Philip Chen and C. Y. Zhang, “Data-intensive applications, challenges, techniques and technologies: A survey on big data,” *Information Sciences*, vol. 275, pp. 314–347, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2014.01.015

[5] B. P. Rimal, E. Choi, and I. Lumb, “A taxonomy and survey of cloud computing systems,” in *Proc. 5th International Joint Conference on INC, IMS and IDC*, Seoul, Korea, Aug. 2009, pp. 44–51.

[6] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. 6th Conference on Symposium on Opearting Systems Design & Implementation*, San Francisco, CA, Dec. 2004, p. 10.

[7] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, “Coded MapReduce,” in *Proc. 53rd Annual Allerton Conference on Communication, Control, and Computing*, Monticello, IL, USA, Sep. 2015, pp. 964–971.

[8] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” in *Proc. IEEE International Symposium on Information Theory*, Barcelona, Spain, Jul. 2016, pp. 1143–1147.

[9] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, “A unified coding framework for distributed computing with straggling servers,” *CoRR*, Sep. 2016. [Online]. Available: http://arxiv.org/abs/1609.01690

[10] A. Severinson, A. Graell i Amat, and E. Rosnes, “Block-diagonal coding for distributed computing with straggling servers,” *CoRR*, May 2017. [Online]. Available: http://arxiv.org/abs/1701.06631

[11] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, “Apache spark: A unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[12] Z. Li, J. Higgins, and M. Clement, “Performance of finite field arithmetic in an elliptic curve cryptosystem,” in *Proc. 9th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Cincinnati, OH, USA, Aug. 2001, pp. 249–256.

[13] N. Balakrishnan, E. Castillo, and J.-M. S. Alegria, *Advances in distribution theory, order statistics, and inference.* Birkhäuser Boston, 2006.

[14] B. C. Arnold, N. Balakrishnan, and H. N. Nagaraja, *A First Course in Order Statistics*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, Sep. 2008.

[15] Z. Zhang, L. Cherkasova, and B. T. Loo, “Performance modeling of MapReduce jobs in heterogeneous cloud environments,” in *Proc. 6th IEEE International Conference on Cloud Computing*, Santa Clara, CA, USA, Jun. 2013, pp. 839–846.

[16] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and A. S. Avestimehr, “Coded TeraSort,” *CoRR*, Feb. 2017. [Online]. Available: http://arxiv.org/abs/1702.04850

[17] J. Dean, “Achieving rapid response times in large online services,” 2012. [Online]. Available: https://research.google.com/people/jeff/latency.html

[18] H. Ishii and R. Tempo, “The PageRank problem, multiagent consensus, and web aggregation: A systems and control viewpoint,” *IEEE Control Systems*, vol. 34, no. 3, pp. 34–53, Jun. 2014.

[19] G. Garrammone, “On decoding complexity of Reed-Solomon codes on the packet erasure channel,” *IEEE Communications Letters*, vol. 17, no. 4, pp. 773–776, Apr. 2013.

[20] A. Severinson, “Coded Computing Tools in Python,” May 2017. [Online]. Available: https://doi.org/10.5281/zenodo.581121

[21] M. Luby, “LT codes,” in *Proc. 43rd Annual IEEE Symposium on Foundations of Computer Science*, Vancouver, BC, Canada, Nov. 2002, pp. 271–280.

[22] A. Shokrollahi, “Raptor codes,” *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2567, Jun. 2006.