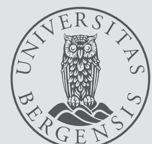
Straggler-Resilient Distributed Computing

Lars Albin Severinson

Thesis for the degree of Philosophiae Doctor (PhD)
University of Bergen, Norway
2022

UNIVERSITY OF BERGEN



Straggler-Resilient Distributed Computing

Lars Albin Severinson

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Date of defense: 09.09.2022

© Copyright Lars Albin Severinson

The material in this publication is covered by the provisions of the Copyright Act.

Year: 2022

Title: Straggler-Resilient Distributed Computing

Name: Lars Albin Severinson

Print: Skipnes Kommunikasjon / University of Bergen

Thanks

Eirik Rosnes and Alexandre Graell i Amat

For teaching me research.

Simula UiB

For making the journey one about more than research.

Abstract

The number and scale of distributed computing systems being built have increased significantly in recent years. Primarily, that is because: i) our computing needs are increasing at a much higher rate than computers are becoming faster, so we need to use more of them to meet demand, and ii) systems that are fundamentally distributed, e.g., because the components that make them up are geographically distributed, are becoming increasingly prevalent. This paradigm shift is the source of many engineering challenges. Among them is the straggler problem, which is a problem caused by latency variations in distributed systems, where faster nodes are held up by slower ones. The straggler problem can significantly impair the effectiveness of distributed systems—a single node experiencing a transient outage (e.g., due to being overloaded) can lock up an entire system.

In this thesis, we consider schemes for making a range of computations resilient against such stragglers, thus allowing a distributed system to proceed in spite of some nodes failing to respond on time. The schemes we propose are tailored for particular computations. We propose schemes designed for distributed matrix-vector multiplication, which is a fundamental operation in many computing applications, distributed machine learning—in the form of a straggler-resilient first-order optimization method—and distributed tracking of a time-varying process (e.g., tracking the location of a set of vehicles for a collision avoidance system). The proposed schemes rely on exploiting redundancy that is either introduced as part of the scheme, or exists naturally in the underlying problem, to compensate for missing results, i.e., they are a form of *forward error correction* for computations. Further, for one of the proposed schemes we exploit redundancy to also improve the effectiveness of multicasting, thus reducing the amount of data that needs to be communicated over the network. Such inter-node communication, like the straggler problem, can significantly limit the effectiveness of distributed systems. For the schemes we propose, we are able to show significant improvements in latency and reliability compared to previous schemes.

List of papers

This thesis is based on the following publications:

Paper I

A. Severinson, A. Graell i Amat, and E. Rosnes, "Block-diagonal and LT codes for distributed computing with straggling servers," *IEEE Transactions on Communications*, vol. 67, no. 3, Mar. 2019.

Paper II

A. Severinson, A. Graell i Amat, E. Rosnes, Francisco Lázaro, and Gianluigi Liva, "A droplet approach based on Raptor codes for distributed computing with straggling servers," in *Proc. International Symposium on Turbo Codes & Iterative Information Processing (ISTC)*, Hong Kong, Dec. 2018.

Paper III

A. Severinson, E. Rosnes, S. El Rouayheb, and A. Graell i Amat, "DSAG: A mixed synchronous-asynchronous iterative method for straggler-resilient learning," under review for potential publication in *IEEE Transactions on Cloud Computing*.

Paper IV

A. Severinson, E. Rosnes, and A. Graell i Amat, "Coded distributed tracking," in *Proc. IEEE Global Communications Conference (GLOBECOM)*, Waikoloa, HI, Dec. 2019.

Paper V

A. Severinson, E. Rosnes, and A. Graell i Amat, "Improving age-of-information in distributed vehicle tracking," in *Proc. URSI General Assembly and Scientific Symposium (GASS)*, Rome, Italy, Aug./Sep. 2021.

Paper I supersedes the following paper, which is not included in this thesis:

A. Severinson, A. Graell i Amat, and E. Rosnes, "Block-diagonal coding for distributed computing with straggling servers," in *Proc. IEEE Information Theory Workshop (ITW)*, Kaohsiung, Taiwan, Feb. 2017.

Contents

Thanks i

Abstract iii

List of papers v

A Introduction 1

1 Background 3
1.1 Introduction . . . 4
1.2 Thesis outline . . . 6
1.3 Thesis scope . . . 6
1.4 Notation . . . 7

2 Distributed systems 9
2.1 Warehouse-scale computers . . . 10
2.1.1 Hardware architecture . . . 10
2.1.2 Hardware failure statistics . . . 11
2.1.3 Software architecture . . . 11
2.2 The straggler problem . . . 15
2.2.1 Round-trip latency on AWS . . . 16
2.2.2 Heterogeneous communication and computation latency on AWS and Azure . . . 17
2.3 Edge computing . . . 19

3 Machine learning and data analytics 23
3.1 Introduction . . . 24
3.2 Linear regression . . . 24
3.3 Logistic regression . . . 26
3.4 Gradient descent . . . 27
3.5 PageRank . . . 29
3.6 The power method . . . 31
3.7 Eigenvalue problems as optimization problems . . . 32
3.8 Principal component analysis . . . 34
3.9 Filtering . . . 36

4 Straggler mitigation 39

4.1 Introduction . . . 40
4.2 Erasure-correcting codes . . . 40
4.3 Coded computing . . . 43
4.4 Gradient codes . . . 46
4.5 Variance-reduced stochastic optimization . . . 47

5 Paper overview **51**

6 Conclusions **55**
6.1 Conclusions . . . 56
6.2 Future work . . . 57

Bibliography **69**

B Papers **71**

I Block-diagonal and LT codes for distributed computing with straggling servers **73**

II A droplet approach based on Raptor codes for distributed computing with straggling servers **93**

III DSAG: A mixed synchronous-asynchronous iterative method for straggler-resilient learning **101**

IV Coded distributed tracking **119**

V Improving age-of-information in distributed vehicle tracking **129**

Part A

Introduction

Chapter 1

Background

1.1 Introduction

Over the past few decades, the amount of data that is stored, processed, and transferred has increased exponentially.[1] Meanwhile, the interest in making sense of and learning from data is also increasing, with a constant stream of novel systems and algorithms being proposed in the machine learning, data analytics, and artificial intelligence communities.[2] However, as a result of increasingly complex algorithms being applied to ever larger datasets, the computational complexity associated with learning from data has increased substantially. For example, the total number of floating-point operations required to train large machine learning models has increased by more than 10 orders of magnitude since the 80s and 90s, see Fig. 1.1. In fact, this trend has been especially strong in recent years—the number of operations required doubled about every 2 years until about 2010 and has been doubling about every 3 to 4 months since then.

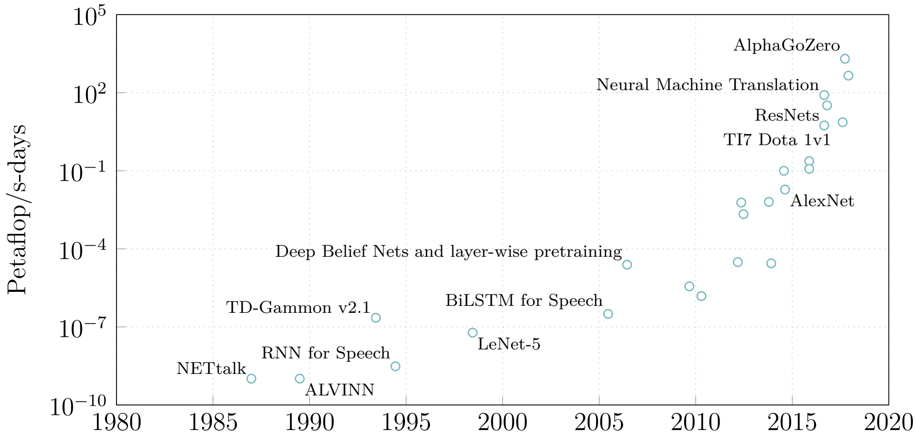


Figure 1.1: Total number of operations required to train large machine learning models, measured in days of computation when performing 10^{15} floating-point operations (i.e., a petaflop) per second. The data is obtained from OpenAI and each marker corresponds to a specific machine learning project; see [3] for details.

Building systems for effectively performing these computations is challenging, and has been the source of a great deal of research in academia and industry [2, 4]. The main takeaway of this research is that dealing with data at this volume has required a fundamental shift in how information technology systems are designed. In particular, we have seen the rise of large-scale distributed systems. These systems are self-healing, largely autonomous, and may span thousands, or even tens of thousands, of servers spread across continents. The reason for this shift is that processors are not becoming faster at nearly the same rate as our computing needs are increasing—single-core processor performance has increased by less than 3 orders of magnitude since the 90s—and that it is becoming increasingly difficult

[1] For example, Cisco estimates that global internet traffic increased by about 7 orders of magnitude between 1992 and 2017 [1].

[2] See, e.g., Chapter 3 and [2, Ch. 2.4] for examples.

to increase processor performance [5]. Instead, we must turn to distributed systems consisting of a large number of processors that cooperatively perform demanding computations. Of particular interest to us are so-called *warehouse-scale computers* (WSCs) [2].

A WSC is a building containing a large number of servers (in some cases tens of thousands [6]) and is similar to traditional data centers.[3] However, WSCs differ from traditional data centers in that all servers are managed through a central entity, referred to as the cluster scheduler (see, e.g., [6–8]), that dynamically assigns compute and storage tasks to servers. These jobs may be very large, potentially spanning all servers that make up the WSC [6, 9]. This is in contrast with traditional data centers, which consist of many small disjoint compute systems that run separate applications. WSCs also differ from so-called high-performance computing (HPC) systems, which are homogenous distributed computing systems built from specialized (and hence expensive) hardware to accelerate certain compute and network operations. In contrast, WSCs are highly heterogeneous and built from commodity hardware to be as cost-effective as possible, at the expense of reliability and predictability. Further, each server in a WSC is typically running several jobs simultaneously to increase utilization, whereas a HPC server typically only runs a single application at a time to ensure that latency and performance is predictable.

As a result of WSCs being a cost-effective way to gain access to large amounts of compute and storage, as well as the recent availability of software to help alleviate the issues inherent to WSCs (e.g., heterogeneity, unreliability, and the sheer number of servers to be managed), many companies now rely on WSCs for their computing and storage needs, often through one of the public cloud offerings, e.g., Google Cloud Platform, Amazon Web Services (AWS), Microsoft Azure, or IBM Cloud.[4] However, there are still several significant challenges associated with WSCs. One such challenge is the so-called *straggler* problem, which is the primary focus of this thesis.

The straggler problem arises in distributed computations where multiple nodes in a network need to cooperate and is a consequence of the heterogeneous and distributed nature of the system. Consider a computation for which a coordinator node needs to aggregate results computed by several worker nodes. Such collective computations are prevalent in distributed systems and are often used, e.g., for machine learning applications, either to increase throughput through parallelization or because the data is distributed. In this case, the coordinator may only be able to proceed once it has received results from all worker nodes, i.e., the overall latency is determined by the slowest node in the network. Further, in large systems the latency of the slowest node may be orders of magnitude higher than the average latency, since the probability that any one node is experiencing a transient outage (e.g., because the network is overloaded) is close to 1 when the number of nodes is large. These slow nodes are known as stragglers, and the severity of the straggler problem increases with the scale and heterogeneity of the system. Hence, it is a

[3]The notion of a WSC as something different from a data center was popularized by engineers working at Google; see [2] for much more on the topic.

[4]For example, Netflix uses AWS for nearly all of its computing and storage needs; see `https://aws.amazon.com/solutions/case-studies/netflix/`.

major issue in WSCs [10]. Further, the problem is even more prevalent in so-called *edge computing systems*, where some nodes are located close to the user (e.g., to provide low latency for certain operations), i.e., at the network edge, and some may be located in a remote WSC. Examples include intelligent transportation systems, where some of the nodes are located at the roadside to facilitate low-latency communication with passing vehicles.

In this thesis, we show how the straggler problem can be alleviated for several important problems through *straggler-resilient* distributed methods of computation. In particular, we propose schemes tailored for matrix-vector multiplication, learning from data, and tracking a time-varying process. These schemes improve speed and reliability by exploiting redundancy to compensate for missing results, i.e., it is a form of *forward error correction* for computations. In addition, one of the schemes tailored for matrix-vector multiplication also exploits redundancy to reduce the amount of data that needs to be moved over the network during the computation, thus further reducing overall latency.[5]

1.2 Thesis outline

This thesis comprises two parts:

- Part 1, which introduces the field and problems considered, and
- Part 2, which is composed of several previously published research papers[6] (i.e., this thesis is an omnibus).

The remainder of Part 1 is organized as follows. In Chapter 2, we cover the anatomy of distributed systems (in particular, WSCs and edge computing systems) and, in Chapter 3, we introduce several important machine learning and data analytics problems. In Chapter 4, we give an overview of two classes of schemes for alleviating the straggler problem for the problems covered in Chapter 3:

- coded computing and
- stochastic optimization.

Finally, in Chapter 5, we give an overview of the papers contained in Part 2, and, in Chapter 6, we draw some conclusions.

1.3 Thesis scope

This thesis is on the topic of straggler-resilient distributed computing schemes. Related topics outside the scope of this thesis include:

[5]Such so-called *data shuffling* may account for a significant portion of the overall latency of distributed computations. For example, for a particular cluster at Facebook, communication accounted for more than 50% of overall latency in 26% of cases and more than 70% of latency in 16% of cases [11].

[6]With one exception, which is under review.

1. Practical studies of distributed computations, e.g., measuring latency variations for different classes of distributed computations and systems.
2. Modelling and analysis of the costs and benefits associated with introducing redundancy to distributed computations.

The work on the first of the above topics is necessary to quantify the extent of the straggler problem under different scenarios—it is the basis of research on the straggler problem. See, e.g., [12–18]. The work on the second of the above topics is necessary to understand the impact of introducing redundancy to distributed computations. In particular, it helps answer the question of which type (e.g., replication or coding) and amount of redundancy to introduce for a particular scenario. See, e.g., [19–22] and references therein for an overview. Once these questions have been answered, this thesis describes several methods for achieving a particular type and level of redundancy for specific classes of computations.

1.4 Notation

Throughout Part 1 of the thesis, we use the following notation:

- Lowercase letters (e.g., x) denote scalars, lowercase bold letters (e.g., $\boldsymbol{x}$) denote vectors, uppercase bold letters (e.g., $\boldsymbol{X}$) denote matrices, and uppercase calligraphic letters (e.g., $\mathcal{X}$) denote sets.
- Vectors are row vectors unless otherwise specified.
- We denote the set of real numbers by $\mathbb{R}$.
- We denote set cardinality by $|\cdot|$.
- We denote vector and matrix transposition by $\cdot^\mathsf{T}$.
- We denote vector inner product by $\langle\cdot,\cdot\rangle$.
- We denote the Frobenius norm by $\|\cdot\|_\mathrm{F}$.

For the papers that make up Part 2, we introduce the notation used on a per-paper basis.

Chapter 2

Distributed systems

2.1 Warehouse-scale computers

In this section, we describe the architecture of a modern WSC and how it gives rise to the straggler problem. Primarily, WSCs are designed to provide large amounts of compute and storage resources at low cost. The hardware and software architectures of WSCs, and the associated issues, are a direct consequence of this goal. In particular, WSCs are built from commodity components and are designed to be easily scalable to very large sizes—the median compute cluster at Google is composed of on the order of 10 000 servers [6]. Cost-effectiveness is paramount, since at the scale of modern compute systems—Google's energy usage suggests that they run close to a million servers in total [23, Ch. 1.1.2]—even a small increase in cost-effectiveness translates to large savings.

2.1.1 Hardware architecture

First, we consider the hardware architecture. A WSC is made up of on the order of thousands of servers that are networked together. Typically, these are commodity servers of a few different types, with, e.g., differing CPUs, amount of memory, and access to hardware for accelerating particular computations (so-called *accelerators*), e.g., graphics processing units. As a result, the time required to perform a particular computation may vary depending on where it is performed. These servers are arranged in racks, each of which contains tens of servers, and are networked together in a heterogeneous manner, typically using Ethernet interconnectcs. This is in contrast with HPC systems, which typically use high-speed (e.g., InfiniBand[1]) interconnects and a relatively uniform network structure. In particular, the network of a WSC is typically a fat tree with 2 or 3 levels [2, Ch. 3].[2] We illustrate such an architecture with 3 levels in Fig. 2.1. Here, the servers within each rack are connected with a so-called *top-of-rack* (ToR) network switch. The racks are in turn arranged in groups, such that the ToR switches of all racks within the same group are connected via a set of group-level network switches. Finally, the group-level switches are connected together via a set of network switches at the topmost layer of the tree.

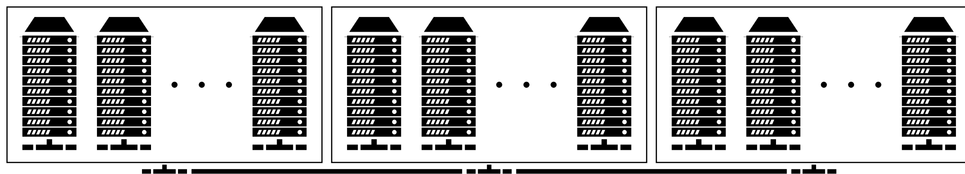


Figure 2.1: A WSC is made up of thousands of commodity servers, potentially equipped with different hardware, that are connected in a heterogeneous manner. Here, servers are arranged in racks, which, in turn, are arranged into groups, with differing numbers of network *hops* between any two servers. As a result computation and communication latency may vary significantly within a WSC.

[1]InfiniBand and other high-speed interconnects can provide latencies orders of magnitude lower than that of Ethernet, but are more expensive.

[2]This is sometimes referred to as a *Clos network*.

The aggregate network bandwidth within a layer of the tree (e.g., between servers in the same rack) is typically much higher than the bandwidth between layers (e.g., between a ToR switch and the group-level switches it is connected to), i.e., the links between layers of the tree are *oversubscribed* [2, Ch. 3]. As a result, communication latency and bandwidth is heterogeneous and may vary significantly depending on where the communicating servers are located within the network, and to what extent other servers utilize the shared communication links.

2.1.2 Hardware failure statistics

As a result of the large number of commodity components that make up a WSC, hardware failure is a common occurrence. For example, the yearly failure statistics of WSCs at Google have been reported as [24]:

- $\approx$ 20 rack failures (40–80 machines disappear for 1–6 hours).
- $\approx$ 5 racks go "wonky" (40–80 servers see $\approx$ 50% packet loss).
- $\approx$ 8 network maintenances ($\approx$ 30-minute random connectivity loss).
- $\approx$ 12 router reloads (transient large-scale network failure).
- $\approx$ 3 router failures (have to immediately reroute traffic for an hour).
- $\approx$ 1000 server failures (out of tens of thousands servers in total).
- $\approx$ 1000s of hard drive failures.

In addition, every second there are 1000s of random delays on the order of milliseconds, and every day there are 10s of random delays on the order of seconds. Removing all variability by using more reliable hardware (if even possible) would be prohibitively expensive. Instead, these issues need to be dealt with in software [2, 10].

2.1.3 Software architecture

Similar to how the operating system of a single computer allows developers to write applications without considering the exact workings of the underlying hardware, WSCs run a software layer that provides a layer of abstraction between the applications of a WSC and the servers that make it up. For the purposes of this thesis, this software layer consists of three main components:

1. Virtualization, for managing the application environment (e.g., dependencies and configuration) and isolating applications sharing the same physical machine.
2. The cluster scheduler, which assigns applications to physical machines depending on resource availability and requirements.
3. Application programming frameworks, that provide high-level APIs for writing applications of specific types.

We go into some detail for each of these components in the following sections.

Virtualization

Virtualization is a set of technologies for creating an isolated *virtual* environment that an application or a set of applications can reside in. In effect, a virtualized environment makes it look to the applications it contains as though there are no other applications sharing the same physical machine. The purpose of this is twofold. First, it ensures that the application sees a consistent environment (e.g., in terms of software dependencies and configuration files) regardless of which physical machine it is deployed to. Second, it keeps applications sharing the same physical machine from interfering with each other by dividing the resources of the physical machine between the virtual environments it hosts.

There are two main categories of virtualization technologies that achieve this goal in slightly different ways:

- *virtual machines* (VMs) and
- *containers.*

VMs virtualize the hardware of the machine, i.e., the VM creates virtual CPUs, memory, etc., and translate calls to the virtual components made by the application running inside the VM into calls to the underlying physical hardware. Examples include QEMU [25], KVM [26], Oracle VirtualBox [27], WMware Workstation [28], VMware EXSi [29], and XEN [30]. Because VMs virtualize the hardware layer, each VM runs its own operating system, which may differ from that of the physical machine acting as the host.

Containers, on the other hand, virtualize operating system resources, e.g., the file system and process identifiers. As a result, despite sharing the same operating system, containers can be isolated from each other. For example, files can be created inside a container without being visible from within other containers. Containers provide a lower level of isolation compared to virtual machines, but do so at a much lower overhead (e.g., in terms of memory usage). As a result, Google runs internal applications inside containers and applications belonging to customers of its public cloud offering, Google Cloud Platform, in virtual machines [6]. Note, however, that even though virtualization technologies limit the resource usage of applications to reduce interference, the behavior of one application may still significantly affect the performance of another due to contention over resources not managed by the VM or container, e.g., memory bandwidth and processor caches [8,31].[3] Further, the cluster scheduler may dynamically reduce the resources available to an application to make those resources available to higher-priority applications. Examples of container technologies include Docker [32] and Open Containers Initiative containers [33], for the Linux kernel, and Jails [34] on the FreeBSD operating system.

WSCs use virtualization extensively since it allows for increasing utilization of scarce hardware resources [8]. In particular, it becomes possible to fully utilize the hardware of a machine by running multiple applications that each utilize a fraction of it, with each application residing in separate VMs or containers. Without virtualization, it would in many cases not be possible to do so, since

[3] This is known as the *noisy neighbor* problem.

applications would interfere with each other, and malicious applications could potentially manipulate or spy on other applications—an especially salient concern in the public cloud (e.g., AWS or Microsoft Azure). In addition, because the VM or container encapsulates any dependencies or configuration required by the application, it becomes possible to run the application on any machine in the WSC, and to easily move it between physical machines. The system for managing this process is referred to as the *cluster scheduler*.

Cluster schedulers

Users of a WSC are not themselves responsible for assigning applications[4] to physical machines. Instead, users submit their applications to a so-called cluster scheduler, which decides where to run each application, accounting for the hardware requirements of the application and the available resources. We illustrate this process in Fig. 2.2. As a result, performance may vary between instances of an application depending on, e.g., if the instance was scheduled on a busy machine or not, and may change over time. In some cases, the scheduler may even preempt (i.e., kill) applications that exceed their resource quota, or make room for other applications with higher priority. For example, AWS provides so-called "Spot Instances", which are VMs that can be preempted by the scheduler at any time, at a discount since it allows them to use spare resources.[5]

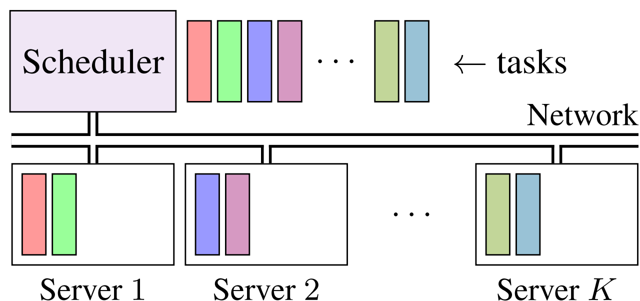


Figure 2.2: A central entity—the cluster scheduler—is responsible for assigning incoming storage and compute tasks (i.e., applications) to servers. Each server is typically assigned several tasks simultaneously to increase utilization, at the expense of increased contention.

Examples of cluster schedulers include Borg [6], Omega [7], and Kubernetes [35], see [8] for an overview. Borg, Omega, and Kubernetes were all started at Google, but while Borg and Omega are internal systems used at Google, Kubernetes is open source and has seen widespread adoption. Another example is the open source Apache Mesos project [36].

[4]Throughout this section, we refer to the VM or container containing an application simply as "the application".

[5]See `https://aws.amazon.com/ec2/spot/`.

Application programming frameworks

To simplify application development, there are several well-known frameworks for expressing specific classes of distributed applications. Perhaps the most well-known example is *MapReduce* [9, 37], where a computation is expressed as a set of map and reduce operations. We illustrate MapReduce with the following example.

Example 1 (MapReduce)**.** *Consider the problem of large-scale matrix-vector multiplication, i.e., the problem of computing*

$$\boldsymbol{y} = \boldsymbol{M}\boldsymbol{x}, \tag{2.1}$$

where $\boldsymbol{M}$ *is a large matrix and* $\boldsymbol{x}$ *is a vector. By partitioning the rows and columns of* $\boldsymbol{M}$ *into an* $N \times M$ *grid, i.e.,*

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{M}_{1,1} & \dots & \boldsymbol{M}_{1,M} \\ \vdots & \ddots & \vdots \\ \boldsymbol{M}_{N,1} & \dots & \boldsymbol{M}_{N,M} \end{bmatrix},$$

and letting

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_N \end{bmatrix},$$

we can equivalently write (2.1) *as*

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}_1 \\ \vdots \\ \boldsymbol{y}_N \end{bmatrix},$$

where

$$\boldsymbol{y}_i \triangleq \sum_{j=1}^{M} \boldsymbol{y}_{i,j} \quad \text{and} \quad \boldsymbol{y}_{i,j} \triangleq \boldsymbol{M}_{i,j}\boldsymbol{x}_i. \tag{2.2}$$

To express a task in the MapReduce framework, one defines the functions map*, which maps an input value to an intermediate key-value pair, and* reduce*, which reduces all key-value pairs with the same key to an output value. As a result, the number of calls to the* map *function is equal to the number of input values and the number of calls to the* reduce *function is equal to the number of unique keys output by the* map *function. To express* (2.1)*, we define*

$$\text{map}(i, j) \to i : \boldsymbol{y}_{i,j} \quad \text{and} \quad \text{reduce}(i) \to \boldsymbol{y}_i,$$

where $\boldsymbol{y}_{i,j}$ *and* $\boldsymbol{y}_i$ *are defined in* (2.2)*. Once defined, the MapReduce framework automatically creates the application instances responsible for running each* map *and* reduce *function, schedules those instances for parallel execution across the WSC, and configures each instance of the* map *function to send its output to the correct instance of the* reduce *function. When each* reduce *function has finished, all vectors* $\boldsymbol{y}_i$ *have been computed.*

One of the main advantages of frameworks such as MapReduce is that they hide much of the complexity associated with writing applications designed to run across potentially thousands of machines. For example, MapReduce automatically handles hardware failures by restarting failed computations [9]. Hence, using the MapReduce framework, large-scale, resilient, distributed applications can be written in a few lines of code. As a result, since the proposal of MapReduce, a range of frameworks have been proposed for various classes of computations. For example, Apache Spark [38, 39] extends the MapReduce model to computations modeled as a directed acyclic graph, i.e., there may be an arbitrary number of stages and each stage may be composed of arbitrary functions. Naiad [40] implements a similar model. Another example is Pregel [41], which is a specialized framework for graph computations. In all cases, the role of the framework is, in large part, to automatically compensate for the hardware failures that inevitably occur in large-scale distributed computing. Further, if the framework is improved (e.g., by adding features to alleviate the straggler problem), all applications expressed in the framework immediately benefit without the applications themselves needing to be updated.

2.2 The straggler problem

WSCs solve a wide range of problems, but also give rise to several challenges. In particular, acheiving low latency and high reliability is difficult. One of the main reasons is the straggler problem, i.e., the problem of some servers (the stragglers) responding more slowly than others. We illustrate the straggler problem in Fig. 2.3. Straggling servers can significantly increase overall latency and reduce resource utilization, since the faster servers need to wait for the stragglers.

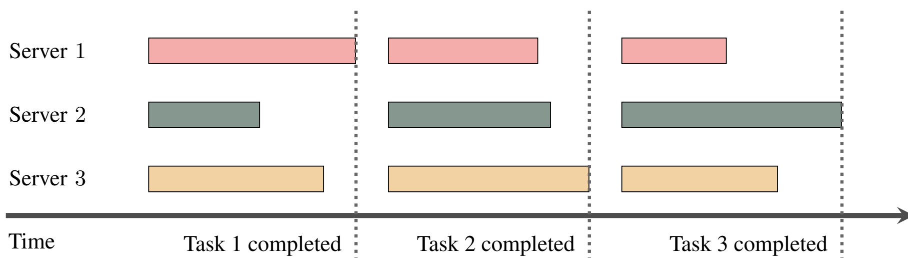


Figure 2.3: The effect of straggling servers on a computation consisting of a sequence of subtasks when distributed over 3 servers; a single slow server (i.e., a straggler) can significantly increase overall latency and decrease hardware utilization.

Fundamentally, the straggler problem is caused by latency variation between servers. Some of the variation is inherent to the components that make up the WSC, but a significant portion of it is due to a combination of

- resource contention and interference from other users and applications, and
- component heterogeneity and unpredictable scheduling.

The straggler problem is an especially salient concern for WSCs (compared to, say, HPC systems) due to the large number of heterogeneous components and due to being designed to be large, cheap, and to maximize resource utilization, at the expense of contention.

As a result, applications running on WSCs may experience large latency variations [10]. For example, for applications scheduled through AWS Lambda,[6] I/O and networking throughput have been observed to vary by as much as a factor 19 when multiple tasks contend for the servers' resources [14]. Further, because the user has no control over scheduling or other users' applications, resource contention is dynamic and unpredictable [15, 16, 42]. In particular, the authors of [16] found that the response time of a web application running in a VM on AWS increased by up to a factor 4 for brief periods of time as a result of contention. On the other hand, some VM types (e.g., AWS `T3` instances[7]) can burst to significantly higher performance for brief periods of time [42]. Further, on Azure Functions (Microsoft's serverless product), a particular task was scheduled on servers with up to 9 different CPU configurations, differing by CPU type and/or number of cores allocated to the task, when called repeatedly [14]. On the other hand, the authors of [17] observed a 25% reduction in performance for a particular web application when it was scheduled on an especially busy server compared to the baseline. These latency variations are largely due to factors outside the control of the user [43].

In the two following sections (Sections 2.2.1 and 2.2.2), we show the impact of stragglers for two specific scenarios for which we have collected latency traces on on AWS and Azure.

2.2.1 Round-trip latency on AWS

Here, we provide latency traces collected on AWS to illustrate the extent to which communication latency varies in WSCs. We conducted the experiment on AWS region `eu-north-1` using two VM instances of type `t2.micro`. Both instances are located in the same WSC. In particular, we measure the latency of

1. sending an ICMP echo probe (i.e., a *ping*) and receiving a response, and
2. transferring a set amount of data between two nodes over a TCP connection.

We use the `fping` utility to record ping latency and the `iperf3` utility to record data transfer latency.[8] We use the client-server mode of `iperf3`, such that one node (the client) establishes a TCP connection with the other node (the server), which runs a daemon that accepts the connection. Once the TCP connection is established, the client node transfers a set amount of data to the server node and records the time until the server node reports having received the data.[9]

[6] AWS Lambda is Amazon's *serverless* product for running applications in lightweight containers, where the AWS cluster manager automatically handles creating application instances.

[7] See `docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html`.

[8] See `https://fping.org` and `https://iperf.fr`, respectively.

[9] We make available the scripts used to record these traces; see `https://github.com/severinson/network-delay-trace`.

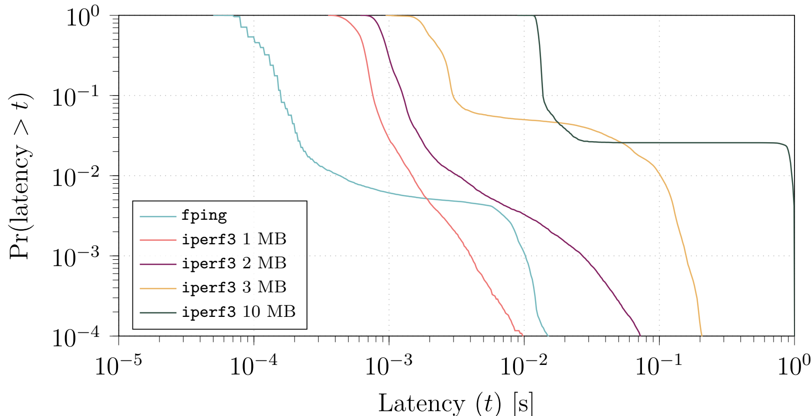


Figure 2.4: CCDF of the round-trip latency measured using the `fping` and `iperf3` utilities between two instances of type `t2.micro` on AWS. The latency of a particular operation can vary by several orders of magnitude.

In Fig. 2.4, we show the empirical complementary cumulative distribution function (CCDF) of the latency associated with these operations. Note that, even for these relatively simple operations, latency may vary by several orders of magnitude. Further, consider how this behavior would affect distributed applications running over a large number of nodes. For example, when waiting to receive 1 MB from a single worker node, the probability of the latency exceeding 10^{-2} seconds is about 10^{-4}. However, with 100 nodes, the probability that all nodes have finished the transfer within 10^{-2} seconds is about 10^{-2}, and with 1000 nodes it is about 10^{-1}.[10] There are two important takeaways. First, it becomes impossible to run latency-critical operations in a distributed environment, such as a WSC, unless steps are taken to mitigate the straggler problem. Second, some of the workers participating in a particular computation may spend most of their time waiting for stragglers, leading to low hardware utilization.

Here, we considered communication latency for two particular worker instances on AWS. In the next section, we consider both communication and computation latency, and we consider to what extent latency varies between workers.

2.2.2 Heterogeneous communication and computation latency on AWS and Azure

In Section 2.2.1, we considered how latency varies for a particular pair of nodes. Here, we quantify the extent to which the mean and variance of the computation and communication latency vary between nodes in a WSC. To do so, we record latency traces for distributed principal components analysis (PCA), which is a particular data analytics computation, on AWS (region `eu-north-1`) and Azure (region `West Europe`), using instances of type `c5.xlarge` and `F2s_v2` for AWS

[10] In fact, it may be even worse since a larger number of nodes using the network simultaneously may increase network contention.

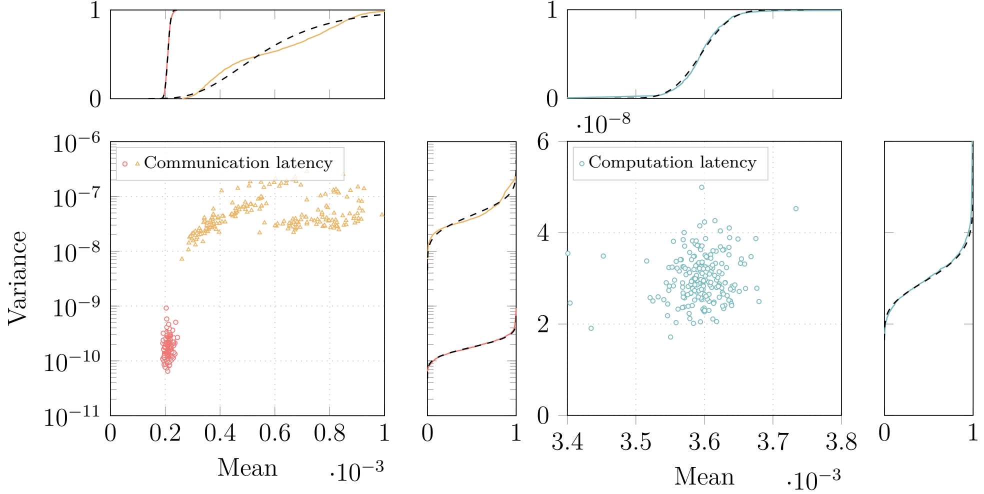


Figure 2.5: Scatter plot of the mean and variance of the per-worker communication (left) and computation (right) latency recorded for 30 048 bytes communicated and $p = 320$ partitions on AWS, and their marginal distributions. For reference, we also plot the CDF of log-normal distributions fitted to the data, shown with black dashed lines. Performance can differ significantly, even between instances of the same type (in this case `c5.xlarge`), i.e., AWS is a heterogeneous platform. In this case, workers come in two classes with different communication speed, which we plot using different colors and markers.

and Azure, respectively. Specifically, we record the latency associated with sending from a coordinator node to a worker node a matrix $\boldsymbol{V}$ and for the worker to respond with the result of the computation

$$\boldsymbol{X}_i^\top \boldsymbol{X}_i \boldsymbol{V}, \tag{2.3}$$

where $\boldsymbol{X}_i$ is a matrix selected at random from the set of matrices that make up a larger data matrix $\boldsymbol{X}$, i.e.,

$$\boldsymbol{X} = [\boldsymbol{X}_1^\top, \dots, \boldsymbol{X}_p^\top]^\top,$$

where p is the number of partitions the rows of $\boldsymbol{X}$ are divided into. The data matrix is derived from the 1000 Genomes phase-3 dataset [44] and is a sparse matrix of size $81\,271\,767 \times 2504$. Further, $\boldsymbol{V}$ is of size 2504×3, and the number of bytes communicated is 30 048 (15 024 bytes in each direction). We give more information on these experiments in Paper III, and the recorded traces and the source code of the software used to record them are available at [45].

We record communication and computation latency separately: the coordinator records the time between sending $\boldsymbol{V}$ to the worker and receiving a response, and the worker records the time between starting to process the received matrix and having a response ready. We take the latency recorded by the worker as a sample of the computation latency, and the difference between the latency recorded by the worker and coordinator as a sample of the communication latency. Hence, we

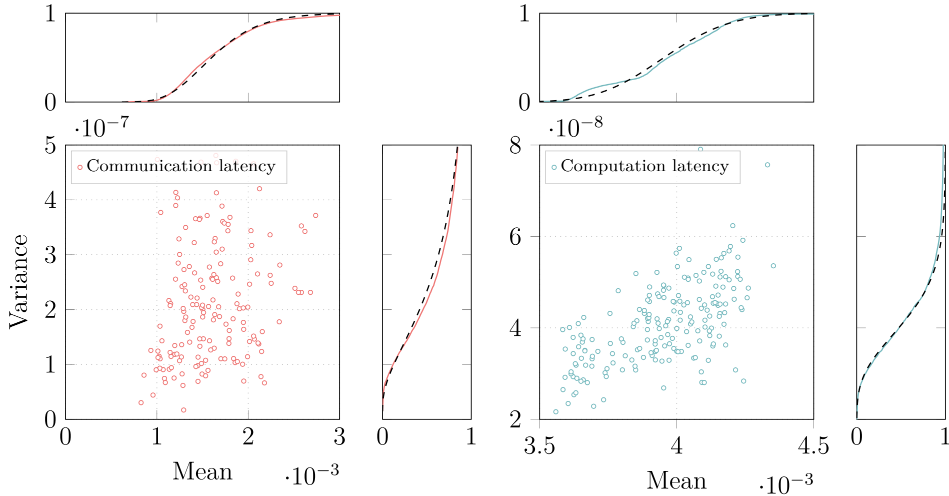


Figure 2.6: Scatter plot of the mean and variance of the per-worker communication (left) and computation (right) latency recorded for 30 048 bytes communicated and $p = 320$ partitions on Azure, and their marginal distributions. Black dashed lines indicate the CDF of log-normal distributions fitted to the data.

record the round-trip communication latency, which includes the time required for data to be sent over the wire and any queuing at either end. For each worker, we perform at least 100 iterations of (2.3).

In Fig. 2.5, we show a scatter plot of the mean and variance of the communication and computation latency recorded for several workers on AWS—each marker corresponds to a worker. We also plot the marginal distributions of the mean and variance, and, for reference, the cumulative distribution function (CDF) of a log-normal distribution fitted to the data. Note that, even though all workers are of the same type and located in the same region, the average communication latency varies by more than a factor 4 between workers. Further, there are two distinct behaviors, which we differentiate by plotting with different colors and markers, perhaps because AWS is using different classes of servers to provide the same instance type (Amazon only specifies an "up to" network speed), or because the instances are located at different places in the WSC (the workers with low communication latency may be located in the same rack as the coordinator). In Fig. 2.6, we show a corresponding plot for Azure. Here, we do not see different classes of workers. However, the amount of variation in computation latency is much greater—up to about 25%.

2.3 Edge computing

WSCs enable a wide range of applications at low cost. However, for devices in remote areas, or for applications requiring low latency, relying on a remote WSC for processing may not be feasible. For example, the 3GPP set out the following

latency requirements as part of the work on 5G [46]:

- Automated cooperative driving: 15 to 75 ms at > 99.9999% (6 nines).
- Factory automation: 0.5 ms at > 99.9999% (6 nines).
- Power plant control: 4 to 16 ms at > 99.999 999 9% (9 nines).

Achieving these goals in the cloud (i.e., while communicating with a remote server on the Internet) would be very challenging. Instead, to achieve stringent latency targets, data has to be processed close to where it is generated, i.e., at the network *edge*, possibly with the support of a remote WSC for operations that are not latency critical. As a result, edge computing is one of the pillars of 5G [47], and several edge computing architectures have been proposed. Among them is multi-access edge computing, proposed by the European Telecommunication Standards Institute, which suggests placing general-purpose servers within the *core network* of mobile network providers, i.e., within the network connecting mobile access points with the Internet, or co-located with the mobile network access points themselves [46–48]. Doing so can reduce latency significantly compared to communicating with a server on the Internet.

To illustrate how the location of a remote server affects latency between the server and a device located at the network edge, in Fig. 2.7 we plot the distribution of the latency associated with:

- Sending a data packet over the air between a wireless modem and an LTE radio access point—separately for uplink (UL) and downlink (DL)—in i) a lab setting, with the parameters of the mobile network tuned to reduce latency (results due to [49]), and ii) in a commercial LTE network deployment (results due to [50]).
- Sending a data packet from an LTE mobile network access point over the wired core network to the point at which the core network connects to the Internet, in the same commercial deployment as referred to in the previous bullet (results due to [50]).
- Setting up a TCP connection between a client with a wired connection to the Internet and a server on the Internet part of the Akamai content delivery network (CDN), which is composed of thousands of servers that are distributed over the world to reduce latency,[11] without transferring any data over the connection (see [51] for details).

We also plot the latency associated with sending a 1-byte message from one Internet-of-things (IoT) device to another using Bluetooth Mesh and OpenThread, which are popular IoT communication protocols (see [52] for details).

As seen, the latency of communicating with a server on the Internet may be prohibitively high—even for this relatively simple operation, the latency exceeds 100 ms in about 20% of the cases. Further, wireless communication between a

[11] See `https://akamai.com/solutions/content-delivery-network` and `https://akamai.com/our-thinking/cdn/what-is-a-cdn`.

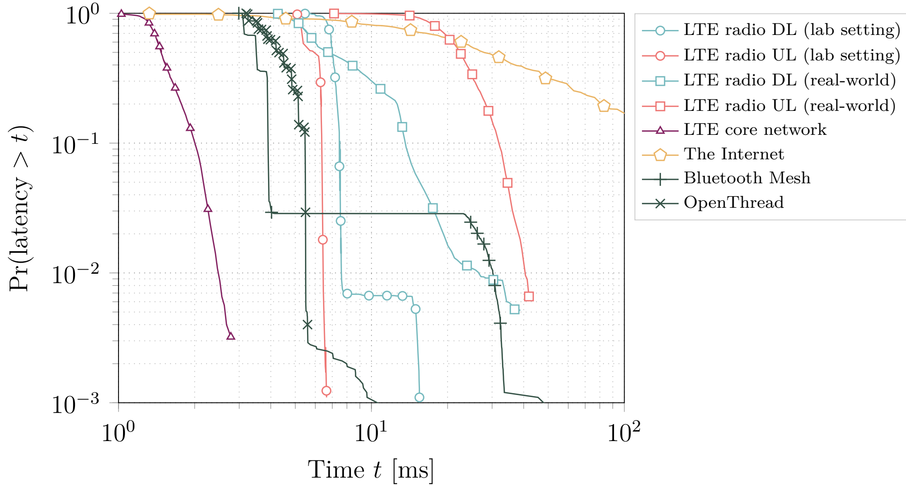


Figure 2.7: Distribution of the latency between a wireless LTE device at the edge and a server co-located with the mobile network base station (results due to [49, 50] for the lab and real-world setting, respectively). We also show the distribution of the latency between the mobile network base station and the point at which the mobile network operator core network connects to the internet Internet (results due to [50]) and between a wired edge device and a server on the Internet (results due to [51]). Finally, we show the latency distribution associated with sending a message consisting of a single byte between two IoT devices using the OpenThread and Bluetooth Mesh communication protocols (results due to [52]). Latency may vary significantly even in controlled settings.

device and a mobile network base station, or even between IoT devices, often requires tens of milliseconds. While it is possible to achieve low latency in mobile networks, mobile networks are often setup to prioritize throughput and fairness, as opposed to latency [49, 50]. Indeed, the lower latency achieved in [49] compared to [50] is largely due to turning off features designed to fairly divide bandwidth between concurrent users.

Hence, even under ideal conditions, to reliably offload processing to remote servers (at the network edge or otherwise), the protocols and algorithms need to be robust against significant latency variations, e.g., by more intelligently allocating scarce resources and utilizing multiple servers and/or communication links simultaneously. This is a relatively new research area in which there is still much work to be done.

Chapter 3

Machine learning and data analytics

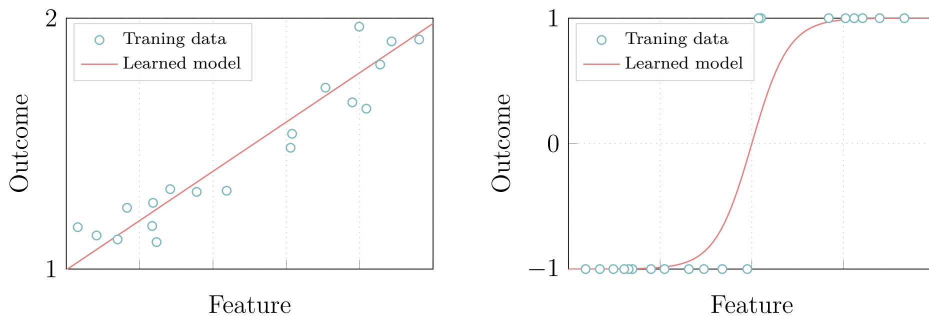


Figure 3.1: Linear (left) and logistic regression (right); we wish to predict an unknown outcome from an observed feature by fitting a line to the training data.

3.1 Introduction

One of the main motivations of today's large-scale distributed computing systems is extracting useful information from large datasets. In this chapter, we introduce several important problems in this domain and explain what fundamental operations they reduce to. In particular, we consider linear and logistic regression, PageRank and PCA, and a filtering problem. Further, we introduce the gradient descent (GD) method for solving optimization problems, the power method for solving eigenvalue problems, such as PageRank and PCA, and show how the power method is a special case of GD. As it turns out, many important problems (e.g., linear and logistic regression, PCA, and training neural networks) reduce to either matrix multiplication or to computing gradients with respect to a dataset (or both). In the next chapter, we will show two classes of schemes for performing these operations in a distributed manner that are resilient to stragglers.

3.2 Linear regression

Consider the problem of predicting the price that a property will be sold for based on features of the property (e.g., its size in square meters and how far it is from the beach). This is known as a *regression problem*, where one wishes to predict an unobserved quantity, referred to as the *outcome*, from one or more observed features, and is a common data analytics problem in a wide range of applications. We first consider *linear regression*, i.e., we constrain the prediction

$$\hat{y}_i = \langle \boldsymbol{x}_i, \boldsymbol{v} \rangle + c$$

to be a linear function of the observed features, where $\boldsymbol{x}_i$ is the vector of features (size and distance to the beach) and $\boldsymbol{v}$ is a vector of length equal to the number of features (2 for the example considered) that, together with an offset c, captures the relationship between the observed features and the quantity to be predicted (the price). The scalar c is commonly referred to as the *intercept*, and the pair $(c, \boldsymbol{v})$ is referred to as the *model*. We illustrate the problem in Fig. 3.1 (left), where we plot the outcome as a function of a single input feature for a sample dataset

(what the feature and dataset represent is not important for the example), where $\boldsymbol{v} = [1]$ and $c = 1$.

Now, the linear regression problem is to find a vector $\boldsymbol{v}$ that minimizes the expected error between the predicted and true outcome, with the error typically measured as the squared difference

$$(\hat{y}_i - y_i)^2 ,$$

where y_i is the true outcome. Because we cannot measure the expected error directly, one typically attempts to find a vector $\boldsymbol{v}$ that minimizes the mean squared error (MSE) between the predicted and true outcomes of a training dataset composed of historical data, for which the true outcomes are known, in the hope that the obtained model will generalize well to new data for which we do not yet know the true outcomes.[1] In this setting, the linear regression problem can be expressed as

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \left[\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \right] = \arg\min_{\boldsymbol{v}} \left[\frac{1}{n} \sum_{i=1}^{n} (\langle \boldsymbol{x}_i, \boldsymbol{v} \rangle + c - y_i)^2 \right] , \tag{3.1}$$

where $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$ and $[y_1, \ldots, y_n]$ are the n feature vectors and corresponding known outcomes that make up the training dataset. Before proceeding, for brevity we move the intercept (c) into the model, such that the prediction of the i-th outcome is written as

$$\hat{y}_i = \langle \boldsymbol{x}_i, \boldsymbol{v} \rangle ,$$

by letting

$$\boldsymbol{x}_i = \left[1, x'_{i,1}, \ldots, x'_{i,n}\right] \quad \text{and} \quad \boldsymbol{v} = [c, v'_1, \ldots, v'_d] ,$$

where $x'_{i,j}$ denotes the j-th entry of the i-th sample and v'_j the j-th entry of the model, prior to moving the intercept into the model.

The expression inside the brackets of (3.1) is referred to as the *loss function*, and the solution to (3.1) minimizes the loss with respect to the training dataset. However, in practice one often uses a slightly updated problem definition

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \left[R(\boldsymbol{v}) + \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \right] , \tag{3.2}$$

where R is a so-called *regularization function* that serves to bias the solution toward ones that are thought to better generalize to data outside of the training dataset, i.e., to reduce *overfitting*, which is the problem of good fit to test data but poor generalization. Regularization can also serve to make the resulting optimization problem easier to solve, and to transform non-convex optimization problems to convex problems. Commonly used regularization functions include the L1 and L2 norm, defined as

$$R(\boldsymbol{v}) = \lambda \sum_{j=1}^{d} |v_i| \quad \text{and} \quad R(\boldsymbol{v}) = \frac{\lambda}{2} \sum_{j=1}^{d} |v_i|^2 ,$$

[1] This approach is known as *empirical risk minimization*.

respectively, where λ is a scalar known as the *regularization coefficient* and d is the length of the vector $\boldsymbol{v}$, i.e., its dimension.[2]

3.3 Logistic regression

So far, we have assumed that the data is of the form

$$y_i = \langle \boldsymbol{x}_i, \boldsymbol{v} \rangle + \epsilon_i, \; i = 1, \ldots, n, \tag{3.3}$$

where ϵ_i is a noise term that accounts for behavior not captured by the model. However, this relation is not always a good fit. For example, see Fig. 3.1 (right), where each outcome takes one of two discrete values in $\{-1, +1\}$. Problems where one whishes to predict outcomes from a discrete set of possibilities are known as *classification problems*, and binary classification problems (i.e., there are two possible outcomes), like the one shown in Fig. 3.1 (right) can be solved with the help of (non-linear) regression.[3] Like with linear regression, we wish to fit a continuous (possibly non-linear) function of the features to the data. Next, to predict a class (either -1 or $+1$), we round the function value to the closest outcome. Further, the difference between the function value and -1 or $+1$ can be interpreted as a measure of how certain the outcome is. For example, a function value of 0.999 could be interpreted as high certainty of $+1$ being the true outcome, whereas a function value close to zero is interpreted as $+1$ and -1 being close to equiprobable.

To address the classification problem, we need a more general relationship between the features and the outcome than the one in (3.3). A popular choice is to model the relationship between features and outcome by

$$y_i = g\left(\langle \boldsymbol{x}_i, \boldsymbol{v} \rangle\right) + \epsilon_i,$$

for some function g with scalar input and output, i.e., the predictions made by the model are of the form $\hat{y}_i = g\left(\langle \boldsymbol{x}_i, \boldsymbol{v} \rangle\right)$, where, as for linear regression, the model is captured by the vector $\boldsymbol{v}$. For classification problems, like the one shown in Fig. 3.1 (right), g is often chosen to be the sigmoid function (scaled and shifted to span $(-1, 1)$), i.e.,

$$g(z) = \frac{2}{1 + \mathrm{e}^{-z}} - 1. \tag{3.4}$$

Here, g serves to map a scalar in $(-\infty, \infty)$ to a value between -1 and $+1$. Next, we need a loss function to asses how well a particular model (captured by $\boldsymbol{v}$) fits the data. A popular choice is to define the problem as

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \left[R(\boldsymbol{v}) + \frac{1}{n} \sum_{i=1}^{n} \log\left[1 + \exp\left(-y_i \langle \boldsymbol{x}_i, \boldsymbol{v} \rangle\right)\right] \right], \tag{3.5}$$

[2] The factor $1/2$ is included to make the gradient of the regularization function simpler to express. Regression with L1-regularization is sometimes referred to as *lasso regression*, whereas regression with L2-regularization is sometimes referred to as *ridge regression*.

[3] There are also extensions for the case where there are more than two possible outcomes, see, e.g., [53, Ch. 5].

in which case the problem is referred to as *logistic regression* (from the logit function, the inverse of the sigmoid function). See [53, Ch. 5] for a motivation of this choice. We plot (3.4) with $z = \langle \boldsymbol{x}, \boldsymbol{v}^* \rangle$ for a sample dataset in Fig. 3.1 (right). Next, we consider how to solve the linear and logistic regression problems.

3.4 Gradient descent

As it turns out, both linear and logistic regression, as well as a wide range of other optimization problems (e.g., training neural networks), can be solved using a simple iterative procedure, in which one makes gradual improvements to an initial guess at the solution. This procedure is referred to as *gradient descent* (GD), and it is used for solving

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \left[F(\boldsymbol{v}) \right], \tag{3.6}$$

for some differentiable function F, by performing updates of the form

$$\boldsymbol{v}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta \nabla F(\boldsymbol{v}^{(t)}),$$

where ∇F denotes the gradient of F, η is the *step size*, which is a scalar that controls the magnitude of the changes, the superscript t is the iteration index, and $\boldsymbol{v}^{(0)}$ is the initial solution (which may be chosen, e.g., at random or set to the all-zeros vector). GD is often likened to descending a mountain in the fog; you take one step at a time in the direction leading downward, without being able to see more than one step ahead, in the hope of eventually reaching the bottom.

In the case of linear and logistic regression, as well as for other optimization problems over a training dataset, the loss function has a finite-sum structure (see (3.2) and (3.5)),[4] i.e.,

$$F(\boldsymbol{v}) = R(\boldsymbol{v}) + \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{v}),$$

where f_i is the loss with respect to the i-th sample of the training dataset. Hence, the gradient of the loss function is given by the sum

$$\nabla F(\boldsymbol{v}) = \nabla R(\boldsymbol{v}) + \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{v}),$$

where ∇f_i denotes the gradient of f_i and ∇R is the gradient of the regularization function. For linear and logistic regression,

$$\nabla f_i(\boldsymbol{v}) = 2 \langle \boldsymbol{x}_i, \boldsymbol{v} \rangle \boldsymbol{x}_i^\top - 2 y_i \boldsymbol{x}_i^\top \quad \text{and} \quad \nabla f_i(\boldsymbol{v}) = \left(g(\langle \boldsymbol{x}_i, \boldsymbol{v} \rangle) - y_i \right) \boldsymbol{x}_i^\top, \tag{3.7}$$

respectively, where g is given by (3.4). We remark that, for linear regression, the gradient can be expressed succinctly as

$$\sum_{i=1}^{n} f_i(\boldsymbol{v}) = 2\boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{v}^\top - 2\boldsymbol{X}^\top \boldsymbol{y},$$

[4]Optimization problems where the loss function has this structure are referred to as *finite-sum* optimization problems.

where

$$\boldsymbol{X} \triangleq \left[\boldsymbol{x}_1^\mathsf{T}, \dots, \boldsymbol{x}_n^\mathsf{T}\right]^\mathsf{T} \quad \text{and} \quad \boldsymbol{y} \triangleq [y_1, \dots, y_n], \tag{3.8}$$

i.e., GD for linear regression can be reduced to matrix-vector multiplication. Further, the gradient of R for L1 and L2 regularization is

$$[\nabla R(\boldsymbol{v})]_j = \lambda \cdot \text{sign}(v_j), j = 1, \dots, d, \quad \text{and} \quad \nabla R(\boldsymbol{v}) = \lambda \boldsymbol{v}, \tag{3.9}$$

respectively.[5]

GD has several advantages that has led to it becoming a popular optimization method for a wide range of problems. Of particular interest to us is that GD is easily distributed by using multiple workers to compute the terms of the sum in (3.7) in parallel, which are then aggregated by a coordinator. Doing so can enable fitting machine learning models over very large datasets quickly.

First-order optimization

GD is part of a larger family of so-called *first-order* iterative optimization methods, referred to as such because they rely only on the gradient of the loss function. These methods differ from each other in two orthogonal ways. First, the methods are either exact, in which case they use the exact gradient (as is the case for GD), or *stochastic*, in which case the method relies on a stochastic approximation of ∇F. A commonly used stochastic method is stochastic GD (SGD), which processes a randomly selected subset of the samples of the training dataset in each iteration,[6] i.e., ∇F is approximated by

$$\nabla \hat{F}(\boldsymbol{v}) = \nabla R(\boldsymbol{v}) + \frac{|\mathcal{I}|}{n} \sum_{i \in \mathcal{I}} \nabla f_i(\boldsymbol{v}), \tag{3.10}$$

where $\mathcal{I}$ is the set of indices of the sampled subgradients. Note that the estimate is scaled by the fraction of samples processed to ensure that it is an unbiased estimate of the gradient, i.e.,

$$\mathbb{E}\left[\nabla \hat{F}(\boldsymbol{v})\right] = \nabla F(\boldsymbol{v}),$$

where the expectation is taken over the possible sets $\mathcal{I}$.[7] We do not estimate ∇R since it is typically easy to compute it exactly. SGD is otherwise equal to GD, i.e., it performs updates of the form

$$\boldsymbol{v}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta \nabla \hat{F}(\boldsymbol{v}^{(t)}),$$

although one typically has to reduce the step size compared to GD, due to the noisy gradient estimate. Other stochastic methods differ by how the gradient is estimated. Stochastic methods can significantly reduce iteration complexity

[5] Note that, for L1-regularization, the gradient is undefined if $v_j = 0$ for some j.

[6] This version of SGD is referred to as mini-batch SGD, to differentiate it from the version of SGD for which exactly 1 sample is processed per iteration.

[7] If the sampling probability is non-uniform, we instead need to scale the loss with respect to each of the sampled indices by the inverse of the associated sampling probability.

relative to GD, which can speed up convergence (when measured as a function of time, as opposed to number of iterations)—it is often better to perform many fast iterations instead of fewer, but more accurate, iterations. However, stochastic methods do not converge to the optimum—there is an irreducible error—unless a variance-reduction strategy is employed, which serves to increase the accuracy of the gradient estimate as the algorithm progresses (see Chapter 4).

The other way in which first-order optimization methods differ is in how they use the gradient (estimated or exact) to update the iterate. One such example is GD with momentum [54, 55], which dampens oscillations in the sequence of iterates $\boldsymbol{v}^{(0)}, \ldots, \boldsymbol{v}^{(t)}$, by including a fraction of the previous update vector in its iterate update. It performs updates of the form

$$\boldsymbol{v}^{(t+1)} = \boldsymbol{v}^{(t)} - \boldsymbol{\theta}^{(t)}, \quad \boldsymbol{\theta}^{(t)} = \gamma\boldsymbol{\theta}^{(t-1)} + \eta\nabla F(\boldsymbol{v}^{(t)}),$$

for some scalar γ (it is common to use $\gamma = 0.9$ or similar), and where $\boldsymbol{\theta}^{(0)}$ is initialized to the all-zeros vector. Other popular examples include accelerated GD [56] and L-BFGS-B [57]. See [58] for a comparison of GD variants. Further, one can mix and match, e.g., combining the SGD gradient estimate with momentum [59], resulting in an update rule

$$\boldsymbol{v}^{(t+)1} = \boldsymbol{v}^{(t)} - \boldsymbol{\theta}^{(t)}, \quad \boldsymbol{\theta}^{(t)} = \gamma\boldsymbol{\theta}^{(t-1)} + \eta\nabla \hat{F}(\boldsymbol{v}^{(t)}),$$

where $\nabla\hat{F}$ is given by (3.10).

Next, we consider eigenvalue problems, which is a another class of important problems in data analytics. However, as we will see, eigenvalue problems can also be cast as optimization problems of the form (3.6) with a loss function of the form (3.7), and be solved using first-order optimization methods, e.g., GD.

3.5 PageRank

One of the most well-known eigenvalue problems is PageRank, which is the problem of computing the relative importance of nodes in a graph based on its network structure. PageRank was originally proposed as an algorithm for ranking webpages by their importance, and has been used by Google to rank search results [60]. Since then, PageRank and variants thereof have been studied extensively, see, e.g., [61,62] and references therein[8], and have seen a wide range of applications, e.g., being used to estimate the importance of research papers, linked by citations [63–66], and to figure out who the best tennis player of all time is [67]. In describing the algorithm, we will use ranking webpages as our running example.

PageRank works as follows. Consider a directed graph with n nodes, each of which corresponds to a webpage, where a directed edge from one node to another corresponds to a hyperlink from the source webpage to the target webpage. The intuition behind PageRank is that webpages with many incoming links from important webpages are themselves important, and that webpages with a small number of incoming links, or webpages only linked to by unimportant webpages, are less important. In particular, PageRank is based on the idea of someone

[8]The explanation of PageRank given here draws primarily from [62].

randomly surfing the web, starting at a randomly selected webpage and repeatedly performing one of the following actions:

1. With probability $1-p$, the surfer follows a link on the current webpage to another webpage.[9] All links on a webpage are assumed to be selected with equal probability.

2. With probability p, the surfer goes directly to another webpage, selected uniformly at random from all nodes in the graph.

The PageRank problem is the problem of, for each webpage, computing the probability that, after k steps, the surfer will be visiting that particular webpage as $k \to \infty$, i.e., to compute the stationary probability distribution over the nodes of the graph. Denote by n_j the number of outgoing hyperlinks from the j-th node and by $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ the *hyperlink matrix*, for which the (i, j)-th entry is

$$a_{i,j} = \begin{cases} 1/n_j & \text{if the } j\text{-th node links to the } i\text{-th node} \\ 0 & \text{otherwise.} \end{cases}$$

For nodes with no outgoing links (e.g., pictures) an artificial link to another node, e.g., one of the pages linking to that page, is added to simulate a back button, so that $n_j > 0, \forall j$. Denote by $\boldsymbol{v}^{(k)} \in \mathbb{R}^n$ the vector storing the probability distribution over the nodes after k steps, i.e., the j-th element of $\boldsymbol{v}^{(k)}$ is the probability of the surfer visiting node j after performing k actions. Note that the process of moving between nodes in the graph is a Markov chain, and that, given the transition matrix of the Markov chain, which we denote by $\boldsymbol{M}$, the probability distribution over the nodes after k state transitions is

$$\boldsymbol{v}^{(k)} = \boldsymbol{M}\boldsymbol{v}^{(k-1)}, \tag{3.11}$$

where $\boldsymbol{v}^{(0)}$ in the initial probability distribution (typically initialized as $v_i = 1/n, \forall i$). The Markov chain transition matrix corresponding to the first action (following a link) is $\boldsymbol{A}$, and the transition matrix corresponding to the second action (moving directly to another webpage) is $1/n \cdot \mathbf{1}_{n\times n}$, where $\mathbf{1}_{n\times n}$ denotes an $n \times n$ all-ones matrix. Hence, the state transition matrix for the overall process is

$$\boldsymbol{M} = (1-p)\boldsymbol{A} + \frac{p}{n}\mathbf{1}_{n\times n}.$$

Now, we are interested in computing the stationary probability distribution, defined as the vector $\boldsymbol{v}^*$ satisfying

$$\boldsymbol{v}^* = \boldsymbol{M}\boldsymbol{v}^*. \tag{3.12}$$

This vector is guaranteed to exist since, because we added artificial links and the surfer sometimes goes directly to another randomly selected page, there are no cycles or sink nodes, i.e., the surfer will never get stuck at any page or in an endlessly repeating cycle. More formally, $\boldsymbol{M}$ is a positive stochastic matrix, i.e., all of its entries are positive and all of its columns sum to 1, which can be shown (by

[9] In the original publication, $p = 0.15$ was suggested [60].

Perron's theorem [68], see [62] for details) to have a unique largest eigenvalue of 1 (i.e., all eigenvalues other than the largest, which is equal to 1, are smaller than 1) and to have a unique eigenvector, for which all entries are positive, corresponding to the eigenvalue 1. Hence, we are guaranteed that $\boldsymbol{v}^*$ exists and is unique since it is, by its definition, the eigenvector of $\boldsymbol{M}$ corresponding to the eigenvalue 1. Next, we consider how to compute $\boldsymbol{v}^*$ using the so-called *power method.*

3.6 The power method

Consider a symmetric positive semidefinite matrix $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ (not to be confused with the regularization coefficient in Section 3.2) and corresponding eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_n$, which, because $\boldsymbol{M}$ is symmetric positive semidefinite, form an orthogonal basis of $\mathbb{R}^n$. Here, we present the power method, which is an iterative method for compting $\boldsymbol{u}_1$ for matrices with a single dominant eigenvalue, i.e., for which the eigenvalues can be ordered such that $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. Let $\boldsymbol{v}^{(0)}$ be an arbitrary vector, which we can write as

$$\boldsymbol{v}^{(0)} = \sum_{i=1}^{n} \alpha_i \boldsymbol{u}_i$$

for some scalars $\alpha_1, \ldots, \alpha_n$. Hence, by multiplying $\boldsymbol{v}^{(0)}$ on the left by $\boldsymbol{M}^k$, we have

$$\boldsymbol{M}^k \boldsymbol{v}^{(0)} = \sum_{i=1}^{n} \lambda_i^k \alpha_i \boldsymbol{u}_i = \alpha_1 \lambda_1^k \left(\boldsymbol{u}_1 + \sum_{i=2}^{n} \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_i}{\lambda_1} \right)^k \boldsymbol{u}_i \right). \tag{3.13}$$

Now, because $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$, the ratio $\left({}^{\lambda_i}/_{\lambda_1}\right)^k \to 0$ and, hence, $\boldsymbol{M}^k \boldsymbol{v}^{(0)} \to \alpha_1 \lambda_1^k \boldsymbol{v}_1$ (assuming $\alpha_1 \neq 0$)[10] as $k \to \infty$, with a rate of convergence approximately proportional to $|\lambda_2|/|\lambda_1|$—eigenvalues smaller than λ_2 typically have negligible impact beyond the first few iterations. While the power method gets its name from (3.13), it is typically not implemented in this way; it is much more computationally efficient to implement the power method recursively as

$$\boldsymbol{v}^{(k)} = \boldsymbol{M}^k \boldsymbol{v}^{(0)} = \boldsymbol{M} \boldsymbol{v}^{(k-1)},$$

i.e., despite its name, the power method is not implemented based on computing powers of $\boldsymbol{M}$. Further, to avoid arithmetic underflow and overflow, the iterate is normalized at each step, i.e.,

$$\boldsymbol{v}^{(k)} = \frac{\boldsymbol{z}^{(k)}}{|\boldsymbol{z}^{(k)}|}, \quad \text{where} \quad \boldsymbol{z}^{(k)} = \boldsymbol{M} \boldsymbol{v}^{(k-1)}.$$

This process is repeated until the series $\boldsymbol{v}^{(0)}, \ldots, \boldsymbol{v}^{(k)}$ satisfies some convergence criterion, e.g., until $|\boldsymbol{v}^{(k)} - \boldsymbol{v}^{(k-1)}|^2$ is smaller than some threshold.

This concludes the description of the power method, which can be used to solve, e.g., the PageRank problem. Like GD, one of its strengths is that it is easily distributed, which can enable solving very large eigenvalue problems.[11]

[10] The probability that $\alpha_1 = 0$ is negligible if $\boldsymbol{v}^{(0)}$ is chosen at random. Further, even if $\boldsymbol{v}^{(0)}$ is deliberately chosen such that $\alpha_1 = 0$, floating-point arithmetic rounding errors quickly introduce a non-zero component in the direction of $\boldsymbol{u}_1$.

[11] For example, the number of nodes in the graph used by Google to rank webpages (and hence the number of rows and columns of the corresponding hyperlink matrix) is in the order of 10^{10} [62].

Computing multiple eigenvectors

While the power method only gives the eigenvector of $\boldsymbol{M}$ corresponding to its dominant eigenvalue, it can be extended to compute the $r \leq n$ eigenvectors corresponding to its r largest eigenvalues. The extension is referred to as the orthogonal power iteration method, and is defined by the update rule

$$\boldsymbol{V}^{(k)} = G\left(\boldsymbol{M}\boldsymbol{V}^{(k-1)}\right),$$

where $\boldsymbol{V}$ is a matrix with r columns and number of rows equal to that of $\boldsymbol{M}$, and $G(\cdot)$ is the Gram-Schmidt operator, i.e., $G(\cdot)$ takes an input matrix and applies the Gram-Schmidt orthogonalization procedure to its columns such that the columns of the resulting matrix form an orthonormal basis with the same span as the columns of the input matrix.[12]

3.7 Eigenvalue problems as optimization problems

Since the PageRank problem reduces to finding an eigenvector of a matrix ((3.12)), we may be interested in casting the problem of computing eigenvectors as an optimization problem, as, in doing so, we can use optimization solvers, e.g., from the family of first-order methods, to solve PageRank (and other eigenvalue problems). In fact, as we will see, the power method is a special case of GD.

Denote by

$$\rho_{\boldsymbol{M}}(\boldsymbol{v}) \triangleq \frac{\langle \boldsymbol{M}\boldsymbol{v}, \boldsymbol{v}\rangle}{\langle \boldsymbol{v}, \boldsymbol{v}\rangle}$$

the Rayleigh quotient of $\boldsymbol{M}$. The Rayleigh quotient has several interesting properties. In particular, it is the scalar closest to an eigenvalue for an arbitrary vector $\boldsymbol{v}$. More precisely,

$$\rho_{\boldsymbol{M}}(\boldsymbol{v}) = \arg\min_{\alpha} |\boldsymbol{M}\boldsymbol{v} - \alpha\boldsymbol{v}|^2,$$

and, in particular, $\rho_{\boldsymbol{M}}(\boldsymbol{u}_i) = \lambda_i$, where λ_i is the i-th largest eigenvalue of $\boldsymbol{M}$ and $\boldsymbol{u}_i$ the corresponding eigenvector. Further, the Rayleigh quotient is a continuos function that attains all values in the range $[\lambda_1, \lambda_n]$ (see, e.g., [69, Sec. 2.7] for proofs), i.e.,

$$\lambda_1 \geq \rho_{\boldsymbol{M}}(\boldsymbol{v}) \geq \lambda_n.$$

Now, let

$$F(\boldsymbol{v}) = R(\boldsymbol{v}) - \frac{\rho_{\boldsymbol{M}}(\boldsymbol{v})}{2} = R(\boldsymbol{v}) - \frac{\langle \boldsymbol{M}\boldsymbol{v}, \boldsymbol{v}\rangle}{2\langle \boldsymbol{v}, \boldsymbol{v}\rangle},$$

and define the optimization problem

$$\begin{aligned} \boldsymbol{v}^* = &\arg\min_{\boldsymbol{v}} \left[F(\boldsymbol{v}) \triangleq R(\boldsymbol{v}) - \frac{\langle \boldsymbol{M}\boldsymbol{v}, \boldsymbol{v}\rangle}{2}\right] \\ \text{s.t.} \quad &|\boldsymbol{v}| = 1, \end{aligned} \tag{3.14}$$

[12]Alternatively, one may use QR factorization instead of Gram-Schmidt, resulting in the QR-method for computing eigenvectors, which typically includes several other optimizations to improve convergence; see [69, Ch. 3] for an overview.

where we have dropped the $\langle \boldsymbol{v}, \boldsymbol{v} \rangle$ in the denominator by constraining $\boldsymbol{v}$ to have unit length, and R is a regularization function. Note that, if R is the identity operator, the solution to (3.14) is the eigenvector of $\boldsymbol{M}$ corresponding to its largest eigenvalue, since the Rayleigh quotient attains its maximum (λ_1) for the input $\boldsymbol{u}_1$.

GD applied to (3.14) results in

$$\boldsymbol{v}^{(k)} = \frac{\boldsymbol{z}^{(k)}}{|\boldsymbol{z}^{(k)}|}, \quad \text{where} \;\; \boldsymbol{z}^{(k)} = \boldsymbol{v}^{(k-1)} - \eta \nabla F\left(\boldsymbol{v}^{(k-1)}\right). \tag{3.15}$$

This is referred to as *projected* GD, since in each iteration we project the iterate onto a subspace, which in this case is the unit ball.[13] Further, the gradient of F is

$$\nabla F(\boldsymbol{v}) = \nabla R(\boldsymbol{v}) - \boldsymbol{M}\boldsymbol{v},$$

and, hence, (3.15) becomes

$$\boldsymbol{z}^{(k)} = \boldsymbol{v}^{(k-1)} - \eta\left(\nabla R(\boldsymbol{v}^{(k-1)}) - \boldsymbol{M}\boldsymbol{v}^{(k-1)}\right).$$

Now, by letting $R(\boldsymbol{z}) = 1/2 \cdot |\boldsymbol{z}|^2$ (i.e., L2-regularization) and choosing the step size to be $\eta = 1$, (3.15) becomes

$$\boldsymbol{z}^{(k)} = \boldsymbol{v}^{(k-1)} - \eta\left(\boldsymbol{v}^{(k-1)} - \boldsymbol{M}\boldsymbol{v}^{(k-1)}\right) = \boldsymbol{M}\boldsymbol{v}^{(k-1)},$$

which is the update rule of the power method, i.e., the power method is a special case of projected GD, for a particular choice of objective, regularization function, and step size. Note that, because we constrain the iterate to have unit norm, L2-regularization only adds a constant to the loss function.[14] This insight is useful, since it allows us to use, e.g., stochastic optimization techniques, or acceleration,[15] in solving eigenvalue problems.

Computing multiple eigenvectors

Similarly, the top $r \leq n$ eigenvectors are given by the solution to

$$\begin{aligned} \boldsymbol{V}^* &= \underset{\boldsymbol{V} \in \mathbb{R}^{n \times r}}{\arg\min} \left[F(\boldsymbol{V}) \triangleq R(\boldsymbol{V}) - \frac{\operatorname{tr}\left(\boldsymbol{V}^\top \boldsymbol{M} \boldsymbol{V}\right)}{2} \right] \\ \text{s.t.} &\quad \boldsymbol{V}^\top \boldsymbol{V} = \boldsymbol{I}, \end{aligned} \tag{3.16}$$

where tr is the trace operator, $\boldsymbol{I}$ is the identity matrix, and the columns of $\boldsymbol{V}^*$ make up the computed eigenvectors.[16] We have

$$\nabla F(\boldsymbol{V}) = \nabla R(\boldsymbol{V}) - \boldsymbol{M}\boldsymbol{V}.$$

[13] In the literature, it is common to consider proximal GD, which allows for introducing a penalty to iterates outside of a particular set. Projected GD is a special case of proximal GD, where the penalty is ∞.

[14] For more information, including a remark on the connection between regularization and the convexity of (3.14), see [70] and [71, Sec. 26.2].

[15] There are even accelerated versions of the power method [72], corresponding to accelerated GD. Further, combining the power method with approximate matrix multiplication methods (e.g., [73]) corresponds to SGD.

[16] See [71, Sec. 26.2] for a remark on the convexity of (3.16).

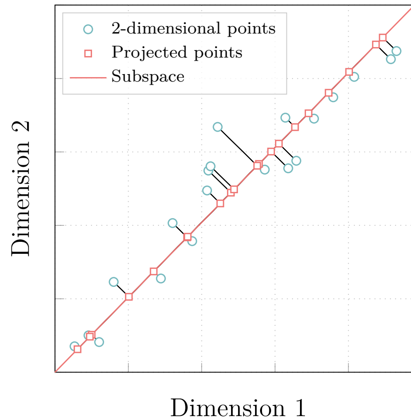


Figure 3.2: PCA; high-dimensional data points are compressed by projecting them onto a subspace. The original points and their projections onto the subspace are connected with black lines.

Thus, by choosing $R(\boldsymbol{V}) = 1/2 \cdot \|\boldsymbol{V}\|_{\mathrm{F}}^2$ (and, hence, $\nabla R(\boldsymbol{V}) = \boldsymbol{V}$) and $\eta = 1$, GD applied to (3.16) results in

$$\boldsymbol{V}^{(k)} = G\left(\boldsymbol{M}\boldsymbol{V}^{(k-1)}\right),$$

where $G(\cdot)$ is the Gram-Schmidt operator, i.e., the orthogonal power iteration method (see Section 3.6), like the power method, is a special case of GD.

3.8 Principal component analysis

PCA relates to dimensionality reduction, i.e., how to best represent a set of points in d dimensions with some lower number of dimensions, here denoted by r ($r < d$). For example, the set of 2-dimensional points shown in Fig. 3.2 could be represented using only 1 coordinate per point if, instead of storing each point as a pair of coordinates (dimension 1 and dimension 2), we represented each point by its closest position along the line shown in the figure, thus reducing the number of coordinates required per point to 1. Note that we also need to store some representation of the line to map points along it back into 2 dimensions. Doing so typically leads to some information loss (the projections do not correspond exactly to the original points), but, for datasets with a high level of redundancy, the loss may be small, and may lead to requiring significantly less storage to represent the dataset. Hence, PCA is often used as a pre-processing step in machine learning.

More formally, PCA is the problem of finding a linear subspace (i.e., a hyperplane) of $\mathbb{R}^d$ of some dimension $r < d$ that best captures a set of n points, denoted by $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$, in the sense that the MSE between the points and their orthogonal projection onto the subspace is minimized. In Fig. 3.2, the subspace is shown as a straight line, with squares indicating the orthogonal projection of the data points (circles) onto the subspace. The model is captured by a matrix $\boldsymbol{V} \in \mathbb{R}^{d \times r}$

with orthonormal columns, i.e., $\boldsymbol{V}^\top \boldsymbol{V} = \boldsymbol{I}$, such that the columns of $\boldsymbol{V}$ form an orthonormal basis of the subspace, and $\boldsymbol{V}\boldsymbol{V}^\top$ is a projection matrix. Hence, the projection of the point $\boldsymbol{x}_i$ onto the subspace is $\boldsymbol{x}_i \boldsymbol{V}\boldsymbol{V}^\top$. The columns of $\boldsymbol{V}$ are referred to as the principal components. The PCA problem is defined as

$$\begin{aligned} \boldsymbol{V}^* = &\operatorname*{arg\,min}_{\boldsymbol{V} \in \mathbb{R}^{d \times r}} \left[F(\boldsymbol{V}) \triangleq \left\| \boldsymbol{X} - \boldsymbol{X}\boldsymbol{V}\boldsymbol{V}^\top \right\|_\mathrm{F}^2 \right], \\ &\text{s.t. } \boldsymbol{V}^\top \boldsymbol{V} = \boldsymbol{I}, \end{aligned} \tag{3.17}$$

where

$$\boldsymbol{X} \triangleq \left[\boldsymbol{x}_1^\top, \dots, \boldsymbol{x}_n^\top \right]^\top .$$

The solution to (3.17) is often expressed in terms of the singular value decomposition (SVD) of the data matrix, which we denote by

$$\boldsymbol{X} = \boldsymbol{W}\boldsymbol{S}\boldsymbol{U}^\top,$$

where the columns of $\boldsymbol{W} \in \mathbb{R}^{d \times k}$, $k = \min(n, d)$, are the left-singular vectors of $\boldsymbol{X}$ (i.e., the eigenvectors of $\boldsymbol{X}\boldsymbol{X}^\top$), the columns of $\boldsymbol{U} \in \mathbb{R}^{d \times k}$ are the right-singular vectors of $\boldsymbol{X}$ (i.e., the eigenvectors of $\boldsymbol{X}^\top \boldsymbol{X}$), and $\boldsymbol{S} \in \mathbb{R}^{k \times k}$ is a diagonal matrix with the singular values $\sigma_1 \geq \cdots \geq \sigma_k$ of $\boldsymbol{X}$ (i.e., the squared eigenvalues of $\boldsymbol{X}\boldsymbol{X}^\top$ and $\boldsymbol{X}^\top \boldsymbol{X}$) arranged along the diagonal. The columns of $\boldsymbol{W}$ and $\boldsymbol{V}$ are mutually orthonormal, i.e., $\boldsymbol{W}^\top \boldsymbol{W} = \boldsymbol{I}$ and $\boldsymbol{V}^\top \boldsymbol{V} = \boldsymbol{I}$.

Denote by $\boldsymbol{w}_i$ and $\boldsymbol{u}_i$ the i-th column of $\boldsymbol{W}$ and $\boldsymbol{U}$, respectively. We can equivalently express the SVD of $\boldsymbol{X}$ as the sum of k rank-1 matrices,

$$\boldsymbol{X} = \sum_{i=1}^{k} \sigma_i \boldsymbol{w}_i \boldsymbol{u}_i^\top = \sum_{i=1}^{k} \boldsymbol{X}\boldsymbol{u}_i \boldsymbol{u}_i^\top = \boldsymbol{X}\boldsymbol{U}\boldsymbol{U}^\top,$$

where we have used the fact that $\boldsymbol{w}_i = {}^1\!/_{\sigma_i} \cdot \boldsymbol{X}\boldsymbol{u}_i$. Hence, $\boldsymbol{X}$ can be approximated by selecting the r ($r \leq k$) first elements of this sum, corresponding to the r largest singular values, i.e.,

$$\boldsymbol{X} \approx \hat{\boldsymbol{X}} \triangleq \sum_{i=1}^{r} \boldsymbol{X}\boldsymbol{u}_i \boldsymbol{u}_i^\top = \boldsymbol{X}\boldsymbol{U}_{1:r}\boldsymbol{U}_{1:r}^\top,$$

where $\boldsymbol{U}_{1:r}$ denotes the matrix consisting of the first r columns of $\boldsymbol{U}$. Further, by the Eckart–Young theorem, $\hat{\boldsymbol{X}}$ is the rank-r matrix closest to $\boldsymbol{X}$ in terms of the Frobenius norm, i.e., the solution to (3.17) is[17]

$$\boldsymbol{V}^* = \boldsymbol{U}_{1:r}.$$

Hence, the PCA problem reduces to the problem of computing the r eigenvectors of $\boldsymbol{X}\boldsymbol{X}^\top$—or $\boldsymbol{X}^\top \boldsymbol{X}$, whichever is easier, since we can recover $\boldsymbol{U}_{1:r}$ from $\boldsymbol{W}_{1:r}$—corresponding to its r largest eigenvalues. As a result, (3.17) can be solved using, e.g., orthogonal power iterations (Section 3.6) or first-order optimization (Section 3.7).

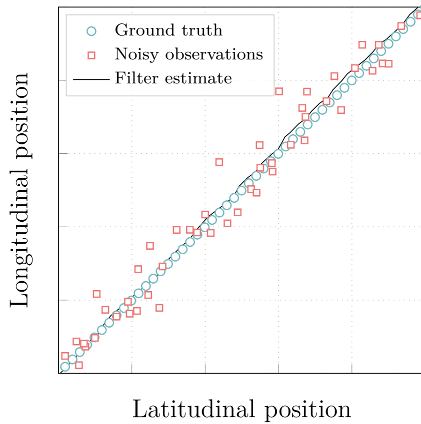


Figure 3.3: Using the Kalman filter to track the longitudinal and latitudinal position and speed of a moving vehicle (we only plot the position component of the estimate). Combining noisy observations of the vehicle's position (e.g., taken using a GPS receiver) with a model of the expected behavior of the system results in higher accuracy compared to relying only on observations.

3.9 Filtering

Finally, we consider a slightly different data analytics problem, namely filtering, i.e., the problem of estimating the state of a process based on (noisy) observations. Throughout this section, all vectors are column vectors. Specifically, we consider a discrete process that evolves over time according to

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{F}\boldsymbol{x}^{(t)} + \boldsymbol{q}^{(t)},$$

where $\boldsymbol{x}^{(t)} \in \mathbb{R}^d$ is a vector that captures the state of the process at time step t, $\boldsymbol{F} \in \mathbb{R}^{d\times d}$ is a model of the expected behavior of the process (referred to as the *state transition matrix*), and $\boldsymbol{q} \in \mathbb{R}^d$ is a noise vector drawn from a zero-mean Gaussian distribution with covariance matrix $\boldsymbol{Q}$. The noise vector accounts for changes to the state of the process not captured by the model. At each time step, we make an observation of the process, which is given as

$$\boldsymbol{z}^{(t)} = \boldsymbol{H}\boldsymbol{x}^{(t)} + \boldsymbol{r}^{(t)},$$

where $\boldsymbol{H} \in \mathbb{R}^{o\times d}$ is a matrix that captures the relationship between the state vector and the observation,[18] and $\boldsymbol{r}$ is zero-mean Gaussian noise with covariance matrix $\boldsymbol{R}$. For example, if we consider the problem of tracking the position and velocity of a vehicle using a GPS receiver, then $\boldsymbol{x}^{(t)}$ could have 4 entries—position and speed in the longitudinal and latitudinal directions—and $\boldsymbol{H}$ could be a matrix

[17]See, e.g., [74] for proofs.
[18]The observation matrix $\boldsymbol{H}$ may be time-varying. We consider a static matrix for notational simplicity.

of size 2×4 that selects only the position elements of $\boldsymbol{x}^{(t)}$, i.e.,

$$\boldsymbol{x}^{(t)} = \begin{bmatrix} \text{lat. position} \\ \text{long. position} \\ \text{lat. speed} \\ \text{long. speed.} \end{bmatrix} \quad \text{and} \quad \boldsymbol{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Now, we wish to estimate the state of the process at each time step, i.e., we wish to produce a sequence of estimates

$$\hat{\boldsymbol{x}}^{(0)},\ \hat{\boldsymbol{x}}^{(1)},\ \hat{\boldsymbol{x}}^{(2)},\ \ldots,$$

such that the MSE is minimized, i.e., we wish to minimize

$$\mathbb{E}\left[\left(\boldsymbol{x}^{(t)} - \hat{\boldsymbol{x}}^{(t)}\right)^2\right].$$

In computing each estimate, we may use all observations captured and state estimates made up to that point.

This problem is solved optimally (i.e., with minimal MSE) by the Kalman filter [75]. At a high level, the Kalman filter is an iterative procedure for producing estimates that performs updates of the form

$$\left(\boldsymbol{x}^{(t-1)}, \boldsymbol{z}^{(t)}\right) \longrightarrow \text{ Kalman filter } \longrightarrow \boldsymbol{x}^{(t)}.$$

Note that the Kalman filter relies only on the current estimate and an observation; all information recorded up to step $t-1$ is captured by the current estimate. The Kalman filter is popular for this reason since it reduces memory usage and simplifies implementation. In addition, the Kalman filter tracks the covariance matrix of the estimation error

$$\boldsymbol{x}^{(t)} - \hat{\boldsymbol{x}}^{(t)},$$

which we denote by $\boldsymbol{P}^{(t)}$. The Kalman filter is used for a wide range of applications, including guidance, navigation and control of vehicles, robotics, and time series analysis, see [76] for an overview.

The Kalman filter works as follows. Let

$$\boldsymbol{y}^{(t)} = \boldsymbol{z}^{(t)} - \boldsymbol{H}\boldsymbol{x}^{(t)}$$

and denote by $\boldsymbol{S}^{(t)} = \boldsymbol{R} + \boldsymbol{H}\boldsymbol{P}^{(t)}\boldsymbol{H}^\intercal$ its covariance matrix. The quantity $\boldsymbol{y}^{(t)}$ is often referred to as the *innovation* and is a measure of how surprising the observation is. Then, the updated state estimate produced by the Kalman filter is

$$\hat{\boldsymbol{x}}^{(t)} = \hat{\boldsymbol{x}}^{(t-1)} + \boldsymbol{K}^{(t)}\boldsymbol{y}^{(t)},$$

where

$$\boldsymbol{K}^{(t)} = \boldsymbol{P}^{(t)}\boldsymbol{H}^\intercal \left(\boldsymbol{S}^{(t)}\right)^{-1}$$

is the so-called Kalman gain that determines how the observation should influence the updated estimate. The covariance matrix of the error $\hat{\boldsymbol{x}}^{(t)} - \boldsymbol{x}^{(t)}$ is also computed recursively as

$$\boldsymbol{P}^{(t)} = \left(\boldsymbol{I}_d - \boldsymbol{K}^{(t)}\boldsymbol{H}\right)\boldsymbol{P}^{(t)},$$

where $\boldsymbol{I}_d$ is the unit matrix of size $d \times d$. We plot the position component of the estimate produced by the Kalman filter for the vehicle tracking problem mentioned above in Fig. 3.3. For more on the mathematics of the Kalman filter, including its derivation, see, e.g., [77].

Chapter 4

Straggler mitigation

4.1 Introduction

In the previous chapter, we outlined several important problems in machine learning and data analytics, and the underlying operations that they rely on—matrix multiplication and computing gradients of finite sums. In this chapter, we give an overview of two sets of methods that have been proposed to perform these operations in a distributed manner that is resilient to delays and *erasures*, that, as we saw in Chapter 2, are common occurrences in large-scale distributed systems. In particular, we consider *coded computing*, which is a method for adding redundancy to computations, such that the result of a distributed computation can be recovered from a subset of the intermediate results computed by the worker nodes, and variance-reduced stochastic optimization. However, we first introduce erasure-correcting codes.

4.2 Erasure-correcting codes

Error-correcting codes were proposed in 1948 to enable reliable communication over unreliable communication channels. More precisely, we wish to send a message through a communication channel (e.g., a wireless link or a fiber-optic cable) that may distort the message in some unknown way (i.e., it is a *noisy* channel), and yet be able to recover the original message at the destination. Error-correcting codes solves this issue by adding redundancy to the message before to sending it over the channel, such that, even if the message is distorted, it can be recovered at the destination. The act of adding redundancy is referred to as *encoding*, and recovering the original message is referred to as *decoding*. At a high level the overall process is

$$\text{source} \xrightarrow[\boldsymbol{m}]{} \text{encoding} \xrightarrow[\boldsymbol{x}]{} \text{noisy channel} \xrightarrow[\hat{\boldsymbol{x}}]{} \text{decoding} \xrightarrow[\hat{\boldsymbol{m}}]{} \text{destination},$$

where $\boldsymbol{m}$ is the original message and $\hat{\boldsymbol{m}}$ is the output of the decoder, which, if decoding is successful, is equal to the original message. Both $\boldsymbol{m}$ and $\hat{\boldsymbol{m}}$ are vectors of length k. Denote by $\boldsymbol{x}$ the channel input (i.e., the encoded message) and by $\hat{\boldsymbol{x}}$ the channel output (i.e., the input to the decoder). We are interested in a particular channel, referred to as the *memoryless erasure channel*, for which the i-th element of the channel output is

$$\hat{x}_i = \begin{cases} x_i & \text{with probability } 1-p \\ \text{“?”} & \text{with probability } p \end{cases}.$$

Here, “?” denotes an *erasure*, indicating that the recipient did not receive x_i, i.e., each element of $\boldsymbol{x}$ is received correctly or not at all, and the recipient knows which elements were received. Each element is erased with probability p, which is referred to as the erasure probability. The erasure channel is often used to model digital communications. Codes designed to protect against erasures (as opposed to errors) are referred to as erasure-correcting codes (ECCs). One of the most common ECCs are repetition codes, which encode a message by repeating it, such that the message can be recovered from any of its replicas. Repetition

codes are widely used due to their simplicity, but are highly suboptimal in terms of the amount of redundancy required. For example, a backup hard disk drive is a 2-repetition code.

In practice, one typically uses linear ECCs, of which repetition codes is a special case, for which an (n, k) code, where k is the number of source symbols (the length of $\boldsymbol{m}$) and n is the length of $\boldsymbol{x}$, is represented by a *generator matrix* of size $n \times k$, denoted by $\boldsymbol{G}$, and the encoded message is

$$\boldsymbol{x} = \boldsymbol{G}\boldsymbol{m}^{\mathsf{T}}.$$

For an (n, k) code, n is referred to as the code length, k as the dimension of the code, and $r = {}^{k}/_{n}$ as the code rate. For example, consider a message $\boldsymbol{m} = [m_1, m_2]$ and a $(3, 2)$ code for which $\boldsymbol{x} = [m_1, m_2, m_1 + m_2]$. In this case, the generator matrix is

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \tag{4.1}$$

and we can recover $\boldsymbol{m}$ from any 2 out of the 3 elements of $\boldsymbol{x}$, i.e., we can tolerate $n - k = 1$ erasure. For linear ECCs, decoding reduces to solving a system of linear equations. Denote by $\boldsymbol{x}_{\mathsf{e}}$ the vector composed of the element of $\hat{\boldsymbol{x}}$ corresponding to the received elements (i.e., those that were not erased) and by $\boldsymbol{G}_{\mathsf{e}}$ the matrix composed of the corresponding rows of $\boldsymbol{G}$. We can recover $\boldsymbol{m}$ from $\hat{\boldsymbol{x}}$, even if some if its elements are erased, by solving the system of equations

$$\boldsymbol{G}_{\mathsf{e}}\boldsymbol{m}^{\mathsf{T}} = \boldsymbol{x}_{\mathsf{e}}. \tag{4.2}$$

For the example in (4.1), if x_1 is erased, the system becomes

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \boldsymbol{m} = \begin{bmatrix} m_2 \\ m_1 + m_2 \end{bmatrix},$$

from which we can solve for $\boldsymbol{m} = [m_1, m_2]$. Note that decoding can only succeed if $\boldsymbol{G}_{\mathsf{e}}$ is of full rank, i.e., if $\operatorname{rank}(\boldsymbol{G}_{\mathsf{e}}) = k$. Decoding that is guaranteed to succeed if $\boldsymbol{G}_{\mathsf{e}}$ is of full rank is referred to as *maximum-likelihood decoding*. For example, Gaussian elimination is a maximum-likelihood decoding algorithm. In some cases, sub-optimal decoding algorithms, which may fail even when $\boldsymbol{G}_{\mathsf{e}}$ is of full rank, are used since they can be made more computationally efficient. A class of ECCs of particular interest is maximum distance separable (MDS) codes, which have the property that the matrix $\boldsymbol{G}_{\mathsf{e}}$ corresponding to any set of at most $n - k$ erasures is of full rank, i.e., $\boldsymbol{G}_{\mathsf{e}}$ is always of full rank if the number of non-erased symbols is at least k. Reed-Solomon codes are a well-known class of MDS codes. Another class of ECCs of particular interest is *fountain codes*.

Fountain codes

Fountain codes (also known as *rateless codes*) are a class of linear ECCs, which, unlike the codes discussed so far (that are referred to as block codes), do not have

a pre-determined code length. Instead, a fountain code can be used to generate an endless stream of coded symbols

$$x_1, x_2, \ldots$$

from a message $\boldsymbol{m}$, each of which is a linear combination of the elements of $\boldsymbol{m}$, i.e.,

$$x_i = \langle \boldsymbol{g}_i, \boldsymbol{m} \rangle$$

for some vector $\boldsymbol{g}_i$. The number of non-zero entries of $\boldsymbol{g}_i$ is referred to as the *degree* of the symbol, and is chosen at random according to a probability distribution over the integers $1, \ldots, k$, which is referred to as the degree distribution of the code. We denote the degree of the i-th symbol by d_i and the degree distribution by Ω. Using a fountain code, each coded symbol is generated as follows:

1. Select the degree of the symbol (d_i) according to the degree distribution Ω.
2. Select d_i source symbols according to some process (e.g., uniformly at random or in a round-robin fashion), which specify the location of the non-zero entries of $\boldsymbol{g}_i$.
3. Populate the d_i non-zero entries of $\boldsymbol{g}_i$ (e.g., by setting them to 1).
4. Compute $x_i = \langle \boldsymbol{g}_i, \boldsymbol{m} \rangle$.

This process can be repeated an indefinite number of times to generate a stream of coded symbols, and can be terminated, e.g., when the recipient signals that decoding has been successful. The recipient can attempt to decode at any time by setting $\boldsymbol{G}_{\mathsf{e}}$ equal to the concatenation of the vectors $\boldsymbol{g}_i$ corresponding to each received coded symbol[1] and attempting to solve the resulting system of linear equations for $\boldsymbol{m}$. Fountain codes are non-MDS, i.e., the recipient must typically collect $k + \delta$ coded symbols before decoding can succeed, for some small δ.

The first practical fountain code to be proposed was Luby Transform (LT) codes [78]. In particular, LT codes combine a computationally efficient, but suboptimal, decoding process known as *peeling decoding* with a degree distribution (known as the Soliton distribution) designed specifically to ensure that the peeling decoding process succeeds with high probability. LT codes have since largely been superseded by Raptor codes [79], which improve upon LT codes by first encoding the message with a high-rate block code (i.e., a code with a pre-determined number of output symbols), and then generating the fountain code symbols from the coded symbols generated by the block code.[2] Raptor codes are designed specifically for a particular decoding algorithm known as inactivation decoding [80],[3] which is an efficient maximum-likelihood decoding algorithm that primarily relies on computationally efficient peeling decoding, but falls back on an optimal decoding process when necessary. There are two standardized versions of Raptor codes

[1]To avoid needing to include the vectors $\boldsymbol{g}_i$ with the message, the sender typically includes a random seed with each coded symbol that allows the recipient to re-create $\boldsymbol{g}_i$ locally.

[2]This is known as a concatenated code.

[3]As part of my work on this thesis, I have developed what I believe to be the fastest open-source implementation of inactivation decoding; see `github.com/severinson/FountainCodes.jl`.

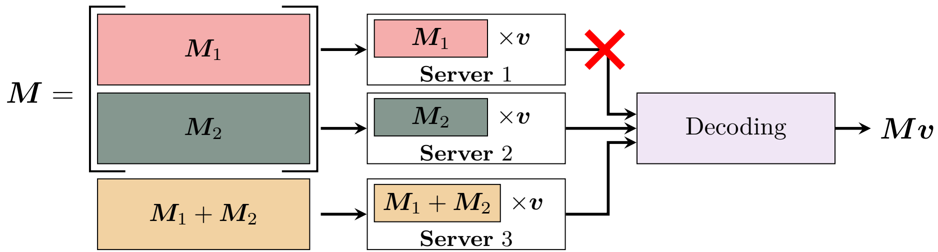


Figure 4.1: Coded distributed matrix multiplication. The results returned from any two workers are sufficient to decode the overall computation output.

that are known as Raptor10 [81] and RaptorQ [82] codes, respectively, both of which are heavily optimized and are close to being MDS in terms of the level of redundancy required. In particular, RaptorQ codes have a greater than 99.9999% probability of decoding success when the number of received symbols is $k + 2$ or greater.

4.3 Coded computing

Coded computing was proposed in 2015 as a method of making distributed computations more resilient through the use of ECCs [83].[4] The idea is to use ECCs to add redundancy to the input of a distributed computation (thus increasing the amount of work assigned to each worker), such that the final result can be recovered from a subset of the intermediate results computed by a set of workers, typically via a decoding operation—like how a message can be recovered even when the data sent over the channel is partly erased. In particular, the authors of [83] showed that coded computing can speed up distributed computations by treating the results computed by straggling workers as erasures, thus obviating the need to wait for the results computed by those workers.

Coded computing schemes are designed with specific computations in mind, and the first scheme to be proposed (in [83]) was designed for matrix-vector multiplication. A later work introduced *gradient codes*, which are designed to recover sums (e.g., the gradient of a loss function with a finite-sum structure) [84]. In addition, many variants of and improvements to these schemes have been proposed. For example, matrix-vector multiplication is considered in [85–89], matrix-matrix multiplication in [90–97], polynomial evaluation in [98], and gradient codes in [84,99–104]. Below, we give an overview of the matrix-vector multiplication scheme of [83] and gradient codes of [84].

Coded matrix-vector multiplication

Coded matrix-vector multiplication is similar to ECCs for communication. We illustrate the scheme with the following example, and, pictorially, in Fig. 4.1.

[4] The paper was first made available in 2015 and was published at a conference in 2016.

Example 2 (Coded matrix-vector multiplication)**.** *Consider the problem of computing the matrix-vector product*

$$\boldsymbol{M}\boldsymbol{v}$$

over a set of $k = 3$ worker nodes. To tolerate any worker not responding, split $\boldsymbol{M}$ row-wise into 2 submatrices, i.e.,

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{M}_1 \\ \boldsymbol{M}_2 \end{bmatrix},$$

and define

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \\ \boldsymbol{X}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}_1 \\ \boldsymbol{M}_2 \\ \boldsymbol{M}_1 + \boldsymbol{M}_2 \end{bmatrix}.$$

Each worker is responsible for computing one of the matrix-vector products

$$\boldsymbol{X}_1\boldsymbol{v}, \ \boldsymbol{X}_2\boldsymbol{v}, \ \text{and} \ \boldsymbol{X}_3\boldsymbol{v}.$$

Now, since

$$\boldsymbol{M}_1\boldsymbol{v} = \boldsymbol{X}_3\boldsymbol{v} - \boldsymbol{X}_2\boldsymbol{v} \ \text{ and } \ \boldsymbol{M}_2\boldsymbol{v} = \boldsymbol{X}_3\boldsymbol{v} - \boldsymbol{X}_1\boldsymbol{v},$$

$\boldsymbol{M}_1\boldsymbol{v}$, $\boldsymbol{M}_2\boldsymbol{v}$, and $\boldsymbol{M}_3\boldsymbol{v}$, and hence $\boldsymbol{M}\boldsymbol{v}$, can be recovered from the results received from any 2 out of 3 workers.

In general, to compute the matrix-vector product $\boldsymbol{M}\boldsymbol{v}$ over a set of n straggling, or otherwise unreliable, workers, such that we can tolerate up to $n - k$ workers not responding, we divide $\boldsymbol{M}$ row-wise into k equally-sized submatrices (possibly after zero-padding $\boldsymbol{M}$ so that the number of rows is divisible by k), i.e.,

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{M}_1 \\ \vdots \\ \boldsymbol{M}_k \end{bmatrix}.$$

Next, encode these submatrices to produce n encoded submatrices, i.e.,

$$\boldsymbol{M}_1, \ldots, \boldsymbol{M}_k \longrightarrow \text{encoding} \longrightarrow \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n.$$

By treating each of $\boldsymbol{M}_1, \ldots, \boldsymbol{M}_k$ as elements of a matrix field, we can express the encoding process as

$$\begin{bmatrix} \boldsymbol{X}_1 \\ \vdots \\ \boldsymbol{X}_n \end{bmatrix} = \boldsymbol{G} \begin{bmatrix} \boldsymbol{M}_1 \\ \vdots \\ \boldsymbol{M}_k \end{bmatrix},$$

where $\boldsymbol{G}$ is a generator matrix of size $n \times k$. Similarly,

$$\begin{bmatrix} \boldsymbol{X}_1\boldsymbol{v} \\ \vdots \\ \boldsymbol{X}_n\boldsymbol{v} \end{bmatrix} = \boldsymbol{G} \begin{bmatrix} \boldsymbol{M}_1\boldsymbol{v} \\ \vdots \\ \boldsymbol{M}_k\boldsymbol{v} \end{bmatrix}. \tag{4.3}$$

Hence, we can recover $\boldsymbol{M}_1\boldsymbol{v}, \ldots, \boldsymbol{M}_k\boldsymbol{v}$ by solving the system of linear equations (4.3), even if some results are not received (i.e., the results are erased),

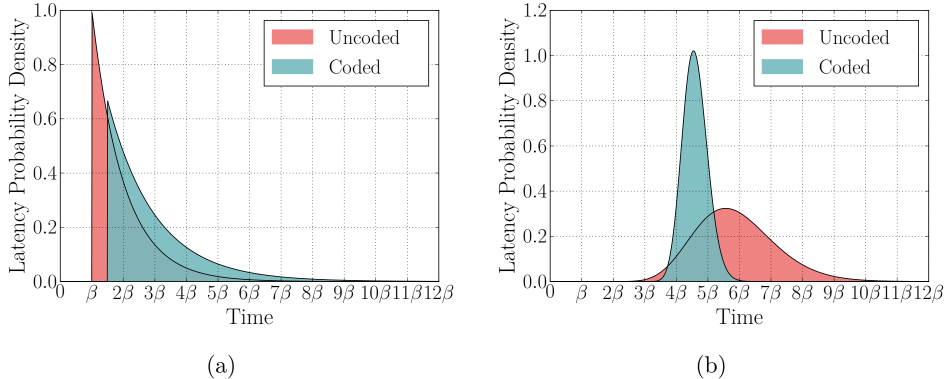


Figure 4.2: Worker subtask latency distribution (a) and latency distribution of the overall computation (b) when distributed over 27 workers. In the coded case, the input matrix is split into 18 submatrices that are encoded using a $(27, 18)$ MDS code, i.e., the final computation output can be decoded from the results returned by any 18 servers. Coding can reduce overall latency despite increasing the latency of each subtask, since the coordinator does not need to wait for straggling workers.

if the matrix $\boldsymbol{G}_{\mathsf{e}}$ composed of the rows of $\boldsymbol{G}$ corresponding to non-erased results is of full rank—just like with linear ECCs used for communication. Coding can thus be used for, e.g., the power method and to compute the gradient of linear regression (which can be expressed as matrix-vector multiplication).

Note that coded matrix-vector multiplication works because the linear relationships between the encoded submatrices introduced by the ECC propagate through the computation, which is also linear.

In the following example, we show how coding can speed up distributed matrix-vector multiplication.

Example 3 (Latency of coded and uncoded matrix-vector multiplication)**.** *In [83], the time taken by each worker to perform its subtask is assumed to be a random quantity that is distributed according to the shifted-exponential distribution, with CDF*

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

where β is a parameter that affects the shift and tail of the distribution. The shift (equal to β) is the minimum amount of time the subtask can be completed in, whereas the tail accounts for transient disturbances, e.g., transmission and queuing delays. The tail of the distribution is the cause of the straggler problem. Here, we set β equal to the number of scalar operations (additions, multiplications, and divisions) required to perform each subtask. Latency is assumed to be independent and identically distributed (i.i.d.) between subtasks.

In Fig. 4.2, we plot the probability density function (PDF) of the latency associated with computing the matrix-vector product $\boldsymbol{Mv}$ over 27 *workers with and without coding. The time axis is scaled by the value of β for the uncoded case. 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* $27/18$ *times more work in the coded case than in the uncoded case. Hence, coding increases subtask latency. However, the expectation of the latency associated with the* 18*-th subtask to be completed in the coded case is lower than that of the* 27*-th subtask to be completed in the uncoded case.*[5] *Coding has thus sped up the computation. Further, the variance of the latency is significantly lower for the coded computation than for the uncoded. Note that we are not accounting for time needed for decoding.*

4.4 Gradient codes

Here, we give an overview of gradient codes, which differ from coded matrix-vector multiplication in that, instead of recovering a set of elements, e.g., $\nabla f_1, \dots, \nabla f_k$, we wish to recover their sum,

$$\nabla f = \sum_{i=1}^{k} \nabla f_i.$$

We show how gradient codes work with the following example.

Example 4 (Gradient codes)**.** *Say that we wish to compute the gradient*

$$\nabla f = \nabla f_1 + \nabla f_2 + \nabla f_3. \tag{4.4}$$

Because computing each of $\nabla f_1, \nabla f_2, \nabla f_3$ *may be computationally expensive, we wish to distribute the work over* $n = 3$ *workers. In the uncoded case, the workers would compute one of* $\nabla f_1, \nabla f_2, \nabla f_3$ *each, and we collect the results at a coordinator responsible for computing their sum. With gradient codes, the three workers would instead compute*

$$1/2 \cdot \nabla f_1 + \nabla f_2, \quad \nabla f_2 - \nabla f_3, \quad \textit{and} \quad 1/2 \cdot \nabla f_1 + \nabla f_3,$$

respectively, i.e., each worker does twice as much work as in the uncoded case. However, we can tolerate any 1 *result being erased (e.g., because the worker assigned to compute it is straggling), since*

- *if the* 1*st result is erased,* $\nabla f = (\nabla f_2 - \nabla f_3) + 2\,(1/2 \cdot \nabla f_1 + \nabla f_3)$,
- *if the* 2*nd result is erased,* $\nabla f = (1/2 \cdot \nabla f_1 + \nabla f_2) + (1/2 \cdot \nabla f_1 + \nabla f_3)$, *and*
- *if the* 3*rd result is erased,* $\nabla f = 2\,(1/2 \cdot \nabla f_1 + \nabla f_2) - (\nabla f_2 - \nabla f_3)$.

In general, gradient codes are described by two matrices $\boldsymbol{A} \in \mathbb{R}^{r \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times k}$, where n is the number of workers, k is the number of partitions the data is divided into, and r is the number of erasure patterns tolerated. In Example 4,

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & -1 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{B} = \begin{bmatrix} 1/2 & 1 & 0 \\ 0 & 1 & -1 \\ 1/2 & 0 & 1 \end{bmatrix}. \tag{4.5}$$

[5] The k-th smallest out of n random variables is known as the k-th order statistic of those random variables. For i.i.d. exponential random variables, the k-th order statistic is a Gamma-distributed random variable; see [105].

Here, $\boldsymbol{B}$ describes what each worker sends to the coordinator. The rows of $\boldsymbol{B}$ correspond to workers and the columns to data partitions, such that the first row of $\boldsymbol{B}$ in (4.5) means that the first worker sends $1/2 \cdot \nabla f_1 + \nabla f_2$ and so on. On the other hand, $\boldsymbol{A}$ describes how to recover the overall sum for a particular erasure pattern. The columns of $\boldsymbol{A}$ correspond to workers and the rows to erasure patterns, with zeros indicating erasures. For example, the first row of $\boldsymbol{A}$ in (4.5) indicates that if the first result is erased, ∇f can be recovered by computing the sum of the second result and the third result multiplied by 2 (i.e., $\nabla f = (\nabla f_2 - \nabla f_3) + 2\,(1/2 \cdot \nabla f_1 + \nabla f_3)$). Hence, for any valid code construction we require that

$$\boldsymbol{A}\boldsymbol{B} = \mathbf{1}_{r\times k}.$$

Two such constructions are proposed in [84] (which one to use depends on n, k, and r). Further, the authors show that, to tolerate that any set of up to s results are erased, each worker must be assigned at least $s+1$ data partitions, i.e., each worker must be assigned $s+1$ times more work compared to the uncoded setting.

4.5 Variance-reduced stochastic optimization

Recall that the learning problems we considered in Chapter 3 (as well as many other problems) can be expressed as a so-called finite-sum optimization problem of the form

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \left[F(\boldsymbol{v}) \triangleq R(\boldsymbol{v}) + \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{v}) \right],$$

and can be solved using first-order iterative optimization methods, for which the computationally expensive step is to compute the gradients

$$\nabla f_1, \ldots, \nabla f_n.$$

One such method is SGD, which performs updates of the form

$$\boldsymbol{v}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta \nabla \hat{F}(\boldsymbol{v}^{(t)}),$$

based on an estimate of the gradient

$$\nabla \hat{F}(\boldsymbol{v}) = \nabla R(\boldsymbol{v}) + \frac{|\mathcal{I}|}{n} \sum_{i\in\mathcal{I}} \nabla f_i(\boldsymbol{v}),$$

where $\mathcal{I}$ is a randomly selected subset of $\{1,\ldots,n\}$. Hence, SGD is naturally straggler-resilient—simply let $\mathcal{I}$ be the indices of the results received from the first workers to respond, and preempt or discard any remaining results. However, there are two issues with this approach, which is known as ignoring-stragglers SGD. First, even at the optimum, the gradient used to update the iterate is noisy, i.e.,

$$\nabla \hat{F}(\boldsymbol{v}^*) \neq \nabla F(\boldsymbol{v}^*).$$

As a result, the method does not converge to the optimum—there is an error floor. Second, if a particular set of workers are straggling over several subsequent

iterations (stragglers tend to remain stragglers), data stored by those workers may never factor into the learning process. This problem is especially severe if the data stored by each worker is not representative of the full dataset, which could be the case, e.g., in a federated learning setting, since data generated by one user is generally not representative of all users.[6] This issue can lead to both a lower rate of convergence and worse performance of the final model. Coded computing was introduced as a way to make distributed computations resilient to stragglers without reducing the quality of the computed result.

However, there are stochastic optimization methods, known as *variance-reduced* methods, for which

$$\nabla \hat{F}\left(\boldsymbol{v}^{(t)}\right) \longrightarrow \nabla F\left(\boldsymbol{v}^{*}\right) \quad \text{as} \quad t \rightarrow \infty,$$

i.e., the gradient estimate tends to the gradient at the optimum as the method progresses—the variance of $\nabla \hat{F}$ is reduced. Informally, the method simultaneously learns the gradient at the optimum and the iterate that minimizes the loss function. As a result, despite relying on a stochastic estimate of the gradient of F, these methods converge to the optimum. The simplest example of a variance-reduced method is SGD with gradually decreasing step size, which can be shown to converge to the optimum [107]. However, a smaller stepsize reduces the rate of convergence, and it is difficult to determine the correct rate at which to reduce the stepsize. Hence, we are interested in variance-reduced methods that can be used with a constant step size, and thus achieve a higher rate of convergence. Methods with this property rely on information contained in previous iterates and/or gradients. Examples include SAG [108], SAGA [109] (including a peer-to-peer version [110]), SARAH [111], SVRG [112], SEGA [113], and MARINA [114]. We compare the rate of convergence of SAG, SGD, and GD for a logistic regression problem in Fig. 4.3. Despite being stochastic, SAG, like GD, but unlike SGD, converges to the optimum. For the remainder of this section, we will briefly describe a family of variance-reduced methods, of which SAG and SAGA are special cases.

Gradient estimation

Consider the problem of estimating the sum

$$\nabla f\left(\boldsymbol{v}^{(t)}\right) \triangleq \frac{1}{n} \sum_{i=1}^{n} \nabla f_i\left(\boldsymbol{v}^{(t)}\right)$$

under the constraint of only being allowed to compute one of the terms

$$\nabla f_1\left(\boldsymbol{v}^{(t)}\right), \ldots, \nabla f_n\left(\boldsymbol{v}^{(t)}\right)$$

at a time. Further, after computing any of these, the iterate is updated, which causes the gradients to change in some unknown way, i.e., $\nabla f_i\left(\boldsymbol{v}^{(t+1)}\right) \neq \nabla f_i\left(\boldsymbol{v}^{(t)}\right)$ in general. The estimate used by SGD is

$$\nabla \hat{f}_{\mathsf{SGD}}^{(t)} = \nabla f_i\left(\boldsymbol{v}^{(t)}\right).$$

[6] As a trivial example, consider learning over the MNIST dataset [106], which consists of handwritten digits 1 to 10, and assigning all samples corresponding to a particular digit to a straggling worker—with ignoring-stragglers SGD, samples of that digit may never factor into the learning process.

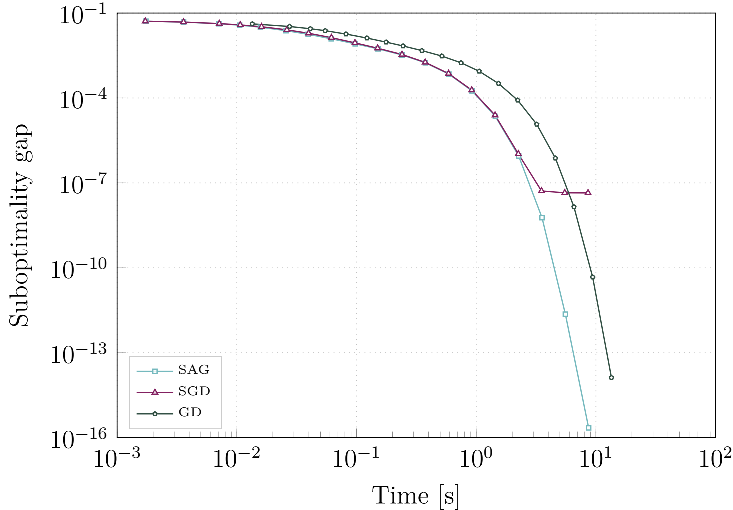


Figure 4.3: The rate of convergence of SAG, which is variance-reduced, SGD, and GD, all with constant step size, for a particular logistic regression problem. On the vertical axis, we plot the suboptimality gap, i.e., the difference in loss between that of the computed iterate and the optimum. For SAG and SGD, we process 10% of the dataset in each iteration. Performing many fast, but inexact iterations, as with SAG and SGD, speeds up initial convergence. However, for SGD, there is an irreducible error due to relying on an estimate of the gradient. SAG, despite being stochastic, converges to the optimum because the variance of its gradient estimate tends to zero. For more details about the problem, see Paper III; the results shown here are those for logistic regression on AWS with $w = 100$ presented there.

Note that, if i is chosen uniformly at random from $\{1, \dots, n\}$,

$$\mathbb{E}\left[\nabla \hat{f}^{(t)}_{\mathsf{SGD}}\right] = \nabla f\left(\boldsymbol{v}^{(t)}\right) \quad \text{and} \quad \mathrm{Var}\left(\nabla \hat{f}^{(t)}_{\mathsf{SGD}}\right) = \mathrm{Var}\left(\nabla f(\boldsymbol{v}^{(t)})\right),$$

where the expectation if taken with respect to the choice of i, i.e., it is an unbiased estimate. However, estimate accuracy does not improve with the number of iterations. SAG [108] and SAGA [109] improve on this estimate by relying on gradients computed in previous iterations. Define[7]

$$\nabla \hat{f}^{(t)}_{\alpha} \triangleq \alpha\left[\nabla f_{i^{(t)}}\left(\boldsymbol{v}^{(t)}\right) - \boldsymbol{z}^{(t-1)}_{i^{(t)}}\right] + \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z}^{(t-1)}_j,$$

where α is a scalar parameter, $i^{(t)}$ is the index of the term computed in the t-th iteration, and $\boldsymbol{z}^{(t)}_1, \dots, \boldsymbol{z}^{(t)}_n$ is a table of previously computed gradients, recursively obtained as

$$\boldsymbol{z}^{(t)}_j = \begin{cases} \nabla f_{i^{(t)}}\left(\boldsymbol{v}^{(t)}\right) & \text{if } j = i^{(t)} \\ \boldsymbol{z}^{(t-1)}_j & \text{otherwise} \end{cases}.$$

[7]This explanation is due to [109].

Hence, $\nabla \hat{f}_\alpha^{(t)}$ is an estimate of $\nabla f(\boldsymbol{v}^{(t)})$ that relies on gradients computed in previous iterations to improve accuracy. There are several strategies for initializing the table entries $\boldsymbol{z}_1^{(0)}, \ldots, \boldsymbol{z}_n^{(0)}$, e.g., $\boldsymbol{z}_i^{(0)} = \mathbf{0}$ or $\boldsymbol{z}_i^{(0)} = \nabla f_i\left(\boldsymbol{v}^{(0)}\right)$ [108, 109].

The expectation and variance of $\nabla \hat{f}_\alpha^{(t)}$ are

$$\mathbb{E}\left[\nabla \hat{f}_\alpha^{(t)}\right] = \alpha\, \mathbb{E}\left[\nabla f\left(\boldsymbol{v}^{(t)}\right)\right] + (1-\alpha)\, \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z}_j^{(t-1)}\right]$$

and

$$\begin{aligned}\operatorname{Var}\left(\nabla \hat{f}_\alpha^{(t)}\right) = \alpha^2 \Bigg[&\operatorname{Var}\left(\nabla f\left(\boldsymbol{v}^{(t)}\right)\right) + \operatorname{Var}\left(\frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z}_j^{(t-1)}\right) \\ &-2\operatorname{Cov}\left(\nabla f\left(\boldsymbol{v}^{(t)}\right), \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z}_j^{(t-1)}\right)\Bigg],\end{aligned}$$

respectively. Hence, the variance is reduced compared to if the covariance between $\nabla \hat{f}_{\mathsf{SGD}}^{(t)}$ and the average of the table entries is large enough—this is true if the step size is small enough, since in that case the gradients do not change too much between iterations. Further, note that by choosing $\alpha = 1$ the estimate is unbiased. In fact, the SAGA optimization method [109] uses exactly this estimate, i.e., $\nabla \hat{f}_{\mathsf{SAGA}}^{(t)} = \nabla \hat{f}_1^{(t)}$. However, by choosing a smaller α, we can further reduce the variance of the estimate at the expense of it becoming biased. Indeed, SAG uses $\alpha = 1/n$, i.e., $\nabla \hat{f}_{\mathsf{SAG}}^{(t)} = \nabla \hat{f}_{1/n}^{(t)}$—in this case the estimate is equal to the sum of the table entries.

We have considered three different estimators (SGD, SAG, SAGA). When used with gradient descent, the iterate update becomes

$$\begin{aligned}
&\text{(SGD)} && \boldsymbol{v}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta \nabla f_{i^{(t)}}\left(\boldsymbol{v}^{(t)}\right),\\
&\text{(SAG)} && \boldsymbol{v}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta \left[\frac{\nabla f_{i^{(t)}}\left(\boldsymbol{v}^{(t)}\right) - \boldsymbol{z}_{i^{(t)}}^{(t)}}{n} + \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z}_j^{(t-1)}\right],\\
&\text{(SAGA)} && \boldsymbol{v}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta \left[\nabla f_{i^{(t)}}\left(\boldsymbol{v}^{(t)}\right) - \boldsymbol{z}_{i^t}^{(t)} + \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{z}_j^{(t-1)}\right],
\end{aligned}$$

where we have omitted the table update. Note that the SAGA estimate is unbiased because it places more weight on the most recently sampled entry, relative to the table entries, compared to SAG.

Chapter 5

Paper overview

Overview

In this chapter, we give an overview of the papers that make up Part 2 of this thesis.

Block-diagonal and LT codes for distributed computing with straggling servers (Paper I)

In Paper I, we consider the problem of multiplying a matrix by a set of vectors. In particular, we propose two schemes that use codes to reduce both computation time and the amount of data that needs to be communicated, when the computation is carried out in a distributed manner over a set of (straggling) servers. These schemes are based on a previous scheme, proposed in [86], which uses MDS codes to achieve these goals. The problem with the approach of [86], which we address in Paper I, is that the encoding and decoding complexity of long MDS codes (i.e., MDS codes with a large number of source and coded symbols) may be prohibitively high. Hence, in the performance results of [86], the time needed for encoding and decoding is not accounted for, despite encoding and decoding being crucial for the scheme to work.

In Paper I, we show that in many scenarios the scheme of [86] results in *increased* overall latency compared to uncoded multiplication when the time needed for encoding and decoding is accounted for. We address this problem by showing how MDS codes can be replaced by LT codes, which is a type of computationally efficient rateless codes, and a special code construction that we propose, referred to as *block-diagonal codes* (BDCs). These codes are non-MDS, i.e., decoding typically requires access to more coded symbols than for an MDS code, which could mean that a larger number of servers must complete their computations before decoding can succeed. However, for BDCs, we show that encoding and decoding complexity can be reduced significantly without increasing the number of servers required. Both schemes of Paper I result in significantly lower latency compared to the scheme of [86], and outperform uncoded multiplication in many cases. To the best of our knowledge, the schemes proposed in Paper I are still among the state-of-the-art for coded matrix-vector multiplication.

A droplet approach based on Raptor codes for distributed computing with straggling servers (Paper II)

One of the principal inefficiencies of coded computing schemes (with a few exceptions) is that they are all-or-nothing, in the sense that either all values computed by a server are utilized, or none are—a server that has completed almost all of its computation contributes no more than a server that has completed none of it. This is a consequence of the decoding stage, which operates over symbols, and of considering the output of a server as a single symbol. In Paper II, we address this limitation by assigning many small computations to each server, the results of which are streamed back to a coordinator (hence, it is a *droplet* approach) responsible for decoding once enough information has been collected. As in Paper I, we consider matrix-vector multiplication. Further, we utilize Raptor codes,

a state-of-the-art rateless code design, which can reduce the overhead required (measured in the number of results collected by the coordinator) before decoding can succeed and improve decoding complexity compared to the schemes in Paper I. Compared to previous schemes in the literature, the scheme proposed in Paper II achieves lower computational delay when the decoding time is taken into account.

DSAG: A mixed synchronous-asynchronous iterative method for straggler-resilient learning (Paper III)

In Papers I and II, we consider matrix-vector multiplication, since it is an important component of many applications, including for many learning problems. In Paper III, we consider learning directly and propose a straggler-resilient first-order optimization method, which is based on asynchronicity and variance-reduction. One of the main advantages of coded computing methods over ignoring-stragglers SGD—which is naturally resilient against stragglers—is that coded computing can achieve straggler-resiliency without any loss in quality, i.e., coded computing methods typically arrive at the same result as a method based on waiting for all servers to complete their computations. Ignoring-stragglers SGD, on the other hand, only produces an approximate result. However, in Paper III, we show that by combining asynchronicity with variance-reduction, it is possible to get the best of both worlds. In particular, we propose a scheme, which we refer to as DSAG, that is naturally resilient against stragglers and that can produce the same result as a synchronous method that waits for all servers to finish their computations, i.e., there is no loss in quality. In addition, DSAG requires no encoding and decoding, instead relying on the redundancy that already exists naturally in large datasets, and on the correlation between gradients computed in subsequent iterations, to compensate for missing results.

The design of DSAG is motivated by behavior observed in cloud computing systems. In particular, we collect latency traces on AWS, Azure, and a local cluster and find that a given worker typically straggles over extended periods of time. This behavior differs from what is often considered in the coded computing literature, where latency is typically modeled as random variables that are identically distributed between workers. We implement DSAG, and we validate its performance by running experiments on clusters composed of up to 100 servers on AWS, Microsoft Azure, and a local cluster. For the scenarios we consider, DSAG is much faster than coded computing and GD, while converging to the same result. We make available the source code and the latency traces we have collected.

Coded distributed tracking (Paper IV)

For Papers I through III, we were primarily interested in learning problems. For the two final papers, Paper IV and Paper V, we consider a different problem: tracking the state of a process that evolves over time in a distributed setting, with multiple observers each observing parts of the state. This is a fundamental information processing problem with a wide range of applications, and in Paper IV we propose a cloud-assisted scheme where the tracking is performed over the

cloud. In particular, to provide timely and accurate updates, and alleviate the straggler problem, we propose a coded distributed computing approach where coded observations are distributed over multiple workers. The proposed scheme is based on a coded version of the Kalman filter that operates on data encoded with an ECC, such that the state can be estimated from partial updates computed by a subset of the workers. We apply the proposed scheme to the problem of tracking multiple vehicles, which is required for, e.g., autonomous driving and collision avoidance systems. We show that replication achieves significantly higher accuracy than the corresponding uncoded scheme. The use of MDS codes further improves accuracy for larger update intervals. In both cases, the proposed scheme approaches the accuracy of an ideal centralized scheme when the update interval is large enough.

Improving age-of-information in distributed vehicle tracking (Paper V)

In Paper V, which is the final paper of the thesis, we consider a particular aspect of the vehicle tracking problem considered in Paper IV: the age-of-information (AoI) of estimate updates. Here, AoI is the time that has passed since the sensor readings that were used to produce the most recent state estimate were taken. Maintaining low AoI is crucial for tracking applications. For example, relying on stale information could result in accidents in a vehicle collision avoidance system. In particular, we derive the AoI of estimate updates and show that replication significantly improves the AoI. Further, we derive the probability that the error of the position estimate exceeds some threshold for a given AoI.

Chapter 6

Conclusions

6.1 Conclusions

Based on our current trajectory, it seems distributed systems will only increase in prevalence, and that, over the coming years, an increasing number of applications will be distributed. Distributed systems are fundamentally different from their centralized counterparts, and many of the (simplifying) assumptions that can be made in designing applications for centralized systems are not valid for distributed systems. In particular, distributed systems are never entirely homogeneous, and are susceptible to many novel *partial* failure scenarios (e.g., the temporary outage of a subset of the nodes that make up the system). This property of distributed systems is the cause of the straggler problem. In contrast, centralized systems are much more homogeneous and typically fail completely when they do fail.

There are two possible approaches for designing reliable and low-latency distributed applications:

1. Designing abstractions that attempt to hide the complexity of the underlying system, such that applications relying on those abstractions can be written under the assumption that no partial failures occur.

2. Designing methods that are inherently tolerant towards partial failures. These methods are to some extent stochastic.

Coded computing (with the exception of approximate coded computing methods), as considered in Papers I and II, is an example of the first approach. This approach significantly simplifies application development. Stochastic optimization methods, including the scheme proposed in Paper III, is an example of the second approach. This approach can result in more performant applications (the scheme of Paper III is much faster than any coded computing method), at the expense of making application development more challenging. The schemes of Papers IV and V can be implemented in either an exact or stochastic manner, i.e., they can be considered examples of either approach.

If the first approach results in a sufficiently performant application (e.g., using coded computing methods for reliable distributed matrix multiplication), it is unnecessary to attempt the second approach, since it may result in a much more complex application. However, in using the first approach, it is important to account for the overhead imposed by the abstraction layer when evaluating application performance. In the case of coded computing methods, this means accounting for the time needed for encoding and decoding, which, paradoxically, is often both significant and ignored in the coded computing literature. In this thesis, we have attempted to rectify this omission. In particular, in Papers I and II, we account for encoding and decoding delay, and proposed coded computing schemes with low encoding and decoding overhead, which can lead to significant performance improvements:

- Paper I: A reduction in computational delay by about a factor 20 (with about a 1% increase in the amount of communication needed) compared to the previous coded computing scheme that the work is based on, for a particular matrix multiplication problem.

- Paper II: An overall delay that is less than half that of the most closely related previous scheme when the number of server is large (on the order of 200). The advantage is smaller than for Paper I since, in Paper II, we consider a matrix multiplication problem for which encoding and decoding accounts for a smaller portion of the overall complexity.

However, the first approach is problematic for two reasons. First, for any particular application, the abstraction layer may take away too much or too little of the complexity of the underlying system. Second, any abstraction invariably *leaks*, i.e., the complexity of the underlying system will shine through for some failure scenario, and the application will misbehave unless designed with that failure scenario in mind.[1] Hence, even when using the first approach, we cannot completely disregard the complexity of the underlying system. Instead, to achieve the highest reliability and performance possible, applications should be designed to do the best they can with the resources available at each point in time. Such applications are necessarily stochastic,[2] but can be very performant. For example, the scheme proposed in Paper III is more than twice as fast as any coded computing method and up to about 50% faster than the most closely related stochastic optimization method for a particular learning problem. The schemes of Papers IV and V lie between the two approaches, in the sense that they achieve performance (as measured by tracking accuracy) close to that of an ideal centralized system up to a point, after which additional failures result in significantly reduced performance.

6.2 Future work

Large-scale distributed systems have only recently become commonplace, and, hence, much work remains in this area and on related topics (see Section 1.3). Here, we give a few ideas.

Joint distributed storage and compute schemes

In WSCs, data is typically stored in a distributed manner and is either replicated or encoded to protect against node or hard drive failures. If data is encoded, it is done so at the byte level, i.e., the linear combination of two pieces of data is the linear combination of the sequences of bytes that make up those pieces, regardless of what those bytes represent. As a result, the first step of coded computing schemes is to read (and decode, if encoded) the data needed for the computation and then encode it in a data-aware manner. For example, if the data being stored consists of 64-bit floating point numbers, the linear combination of two pieces of data is the linear combination of those 64-bit floating point numbers. This is a time-consuming process that may require moving data across the network. Hence, we suggest researching joint storage and compute schemes, for which data-aware codes are used for reliable storage, such that coded computing can be performed

[1] See `https://joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/` for more on the topic.

[2] To see why, consider a distributed system that produces an exact answer if no error occurs and “?” otherwise. This system is stochastic, since both outcomes occur with some probability.

directly on the data stored on disk, without needing to re-encode it. It may even be possible to store the coded output directly, without needing to immediately decode it.

Distributed coordination

For both of the two approaches explained in the previous section, most proposed schemes are based around the idea of a single coordinator node responsible for aggregation, which has to always be available. Hence, the coordinator is a single point of failure. Further, as systems grow, the speed at which the coordinator is able to process results computed by the workers will at some point become the limiting factor. For these reasons, it becomes important to develop ways of distributing the work of the coordinator node (we do this to an extent in Papers IV and V). For coded computing, this involves distributing the work of decoding, and, likely, the design of new codes. For distributed learning, it involves designing methods of exchanging information between coordinator nodes that may have received different sequences of results from the workers. Such distributed coordination schemes sit somewhere between schemes with a central coordinator and peer-to-peer systems, where all nodes are tasked with producing an end result.

Coded computing and sketch-and-project

Another exciting topic is sketch-and-project methods, i.e., methods for approximating a quantity from one or more rank-deficit sketches, and their relationship with linear codes. For example, a sketch of a vector is the product of a rank-deficit sketching matrix and the vector. The sketch can be used to improve an estimate by projecting the current estimate onto the set of vectors consistent with the sketch.[3]

In particular, for encoding, each coded symbol can be considered a sketch, and projection reduces to decoding (i.e., solving a system of linear equations) if the sketches combined have full rank. Hence, sketch-and-project gives an answer for how a system should behave if decoding fails, thus making it possible for coded computing methods to give an approximate result.

Applications and understanding of stochastic optimization

Many promising stochastic optimization methods have been proposed in the mathematical optimization literature recently (e.g., [114]) that could potentially be used as the basis for straggler-resilient distributed computing schemes (as we did in Paper III). However, much work remains on designing, implementing, and understanding those schemes. Especially since, in designing these schemes, we will need to understand how these stochastic methods behave under a wide range of partial failure scenarios.

The principal challenge of stochastic methods may be in understanding their performance characteristics and how to tune them for a particular scenario. For example, it is often not clear what guarantees they offer, and the analysis that

[3] The Kalman filter can be interpreted as projection.

does exist is often overly pessimistic, in the sense that actual performance (as measured, e.g., in convergence rate for a learning problem) may be much higher than the analysis would indicate. Developing a more precise understanding will be important for applications where we require some guarantee on the accuracy of the computed result.

Bibliography

[1] Cisco, "Cisco visual networking index: Forecast and trends, 2017–2022," Cisco, Tech. Rep., 2019.

[2] L. A. Barroso, U. Hölzle, P. Ranganathan, and M. Martonosi, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed. Morgan & Claypool Publishers, 2018.

[3] D. Amodei, D. Hernandez, G. Sastry, J. Clark, G. Brockman, and I. Sutskever. (2018) AI and compute. [Online]. Available: https://openai.com/blog/ai-and-compute/

[4] M. Kleppmann, *Designing Data-Intensive Applications.* O'Reilly Media, 2017.

[5] H. Khan, D. Hounshell, and E. Fuchs, "Science and research policy at the end of Moore's law," *Nature Electron.*, vol. 1, no. 1, pp. 14–21, Jan. 2018.

[6] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, Bordeaux, France, Apr. 2015.

[7] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, Prague, Czech Republic, Apr. 2013, pp. 351–364.

[8] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[9] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, San Francisco, CA, Dec. 2004, pp. 137–149.

[10] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[11] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," in *Proc. ACM Special Interest Group Data Comm. Conf. (SIGCOMM)*, Toronto, ON, Canada, Aug. 2011, pp. 98–109.

[12] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Lombard, IL, Apr. 2013, pp. 185–198.

[13] G. Liang and U. C. Kozat, "TOFEC: Achieving optimal throughput-delay trade-off of cloud storage using erasure codes," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Toronto, ON, Canada, Apr. 2014, pp. 826–834.

[14] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annual Tech. Conf. (ATC)*, Boston, MA, Jul. 2018, pp. 133–146.

[15] A. Maji, S. Mitra, and S. Bagchi, "ICE: An integrated configuration engine for interference mitigation in cloud services," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Grenoble, France, Jul. 2015, pp. 91–100.

[16] S. A. Javadi and A. Gandhi, "DIAL: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Columbus, OH, Jul. 2017, pp. 135–144.

[17] X. Han, R. Schooley, D. Mackenzie, O. David, and W. J. Lloyd, "Characterizing public cloud resource contention to support virtual machine co-residency prediction," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Sydney, Australia, Apr. 2020, pp. 162–172.

[18] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Intel Science & Technology Center for Cloud Computing, Carnegie Mellon University, Tech. Rep. ISTC-CC-TR-12-101, 2012.

[19] K. Gardner, M. Harchol-Balter, A. Scheller-Wolf, and B. Van Houdt, "A better model for job redundancy: Decoupling server slowdown and job size," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3353–3367, Dec. 2017.

[20] K. Gardner, M. Harchol-Balter, A. Scheller-Wolf, M. Velednitsky, and S. Zbarsky, "Redundancy-d: The power of d choices for redundancy," *Operations Research*, vol. 65, no. 4, pp. 1078–1094, Aug. 2017.

[21] M. F. Aktaş and E. Soljanin, "Straggler mitigation at scale," *IEEE/ACM Trans. Netw.*, vol. 27, no. 6, pp. 2266–2279, Dec. 2019.

[22] U. Ayesta, T. Bodas, and I. M. Verloop, "On a unifying product form framework for redundancy models," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 46, no. 3, pp. 80–81, Dec. 2018.

[23] M. Luksa, *Kubernetes in Action.* Manning, 2017.

[24] J. Dean, "Building software systems at Google and lessons learned," *Stanford Computer Science Department Distinguished Computer Scientist Lecture*, Nov. 2010. [Online]. Available: http://research.google.com/people/jeff/Stanford-DL-Nov-2010.pdf

[25] QEMU Contributors. (2021) QEMU. [Online]. Available: qemu.org

[26] KVM Contributors. (2021) KVM. [Online]. Available: linux-kvm.org/

[27] Oracle Corporation. (2021) VirtualBox. [Online]. Available: virtualbox.org

[28] WMware. (2021) WMware Workstation Pro. [Online]. Available: vmware.com/no/products/workstation-pro.html

[29] ——. (2021) WMware ESXi. [Online]. Available: vmware.com/no/products/esxi-and-esx.html

[30] The Linux Foundation. (2021) XEN project. [Online]. Available: xenproject.org/

[31] J. Langguth, X. Cai, and M. Sourouri, "Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures," in *Proc. IEEE Int. Conf. Parallel Dist. Syst. (ICPADS)*, Singapore, Dec. 2018, pp. 497–506.

[32] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[33] The Linux Foundation. (2021) Open container initiative. [Online]. Available: opencontainers.org/

[34] The FreeBSD Project. (2021) Jails. [Online]. Available: docs.freebsd.org/en/books/handbook/jails/

[35] The Kubernetes Authors. (2021) Kubernetes. [Online]. Available: kubernetes.io

[36] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, Mar. 2011, pp. 295–308.

[37] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symp. Mass Storage Syst. Tech. (MSST)*, Incline Village, NV, May 2010.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. USENIX Conf. Hot Topics Cloud Comput. (HotCloud)*, Boston, MA, Jun. 2010.

[39] 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 *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, San Jose, CA, Apr. 2012, pp. 15–28.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. ACM Symp. Operating Syst. Principles (SOSP)*, Farminton, PA, Nov. 2013, pp. 439–455.

[41] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM Int. Conf. Management Data (SIGMOD/PODS)*, Indianapolis, IN, Jun. 2010, pp. 135–146.

[42] "Managing resource bursting," U.S. Patent 9,417,902, Aug., 2016.

[43] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, Y. Xiong, and S. Dustdar, "SLOC: Service level objectives for next generation cloud computing," *IEEE Internet Comput.*, vol. 24, no. 3, pp. 39–50, May 2020.

[44] A. Auton *et al.*, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, pp. 68–74, Oct. 2015.

[45] A. Severinson. (2021) DSAG source code and latency traces. [Online]. Available: https://github.com/severinson/DSAG-Paper

[46] 3GPP, "Study on communication for automation in vertical domains," 3GPP, Tech. Rep. 22.804 V16.3.0, Jul. 2020.

[47] S. Kekki, W. Featherstone, Y. Fang, P. Kuure, A. Li, A. Ranjan, D. Purkayastha, F. Jiangping, D. Frydman, G. Verin, K.-W. Wen, K. Kim, R. Arora, A. Odgers, L. M. Contreras, and S. Scarpina, "MEC in 5G networks," ETSI, Tech. Rep. 28, Jun. 2018.

[48] 3GPP, "Architecture for enabling edge applications," 3GPP, Tech. Rep. 23.558 V17.1.0, Sep. 2021.

[49] G. Pocovi, I. Thibault, T. Kolding, M. Lauridsen, R. Canolli, N. Edwards, and D. Lister, "On the suitability of LTE air interface for reliable low-latency applications," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Marrakesh, Morocco, Apr. 2019.

[50] M. Laner, P. Svoboda, P. Romirer-Maierhofer, N. Nikaein, F. Ricciato, and M. Rupp, "A comparison between one-way delays in operating HSPA and LTE networks," in *Proc. IEEE Int. Symp. Modeling Opt. Mobile, Ad Hoc Wireless Netw. (WiOpt)*, Paderborn, Germany, Aug. 2012.

[51] R. Al-Dalky and M. Rabinovich, "Revisiting comparative performance of DNS resolvers in the IPv6 and ECS era," Jul. 2020. [Online]. Available: https://arxiv.org/abs/2007.00651

[52] J. Algrøy, "Latency in mesh networks," Nov. 2021. [Online]. Available: https://arxiv.org/abs/2201.03470

[53] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed. Unpublished, 2021, see https://web.stanford.edu/~jurafsky/slp3/.

[54] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Comput. Math. Math. Physics*, vol. 4, no. 5, 1964.

[55] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Netw.*, vol. 12, no. 1, pp. 145–151, Jan. 1999.

[56] Y. E. Nesterov, "A method of solving a convex programming problem with convergence rate $o\left(\frac{1}{k^2}\right)$," *Soviet Math. Doklady*, vol. 269, no. 3, pp. 543–547, 1983.

[57] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, "L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization," *ACM Trans. Math. Softw.*, vol. 23, no. 4, pp. 550–560, Dec. 1997.

[58] S. Ruder, "An overview of gradient descent optimization algorithms," Sep. 2016. [Online]. Available: https://arxiv.org/abs/1609.04747

[59] Y. Liu, Y. Gao, and W. Yin, "An improved analysis of stochastic gradient descent with momentum," in *Proc. Neural Inf. Process. Syst. (NeurIPS)*, Jun. 2020, pp. 18 261–18 271.

[60] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw.*, vol. 30, pp. 107–117, 1998.

[61] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, 2006.

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

[63] L. Yao, T. Wei, A. Zeng, Y. Fan, and Z. Di, "Ranking scientific publications: the effect of nonlinearity," *Nature Scientific Reports*, vol. 4, Oct. 2014. [Online]. Available: http://www.nature.com/articles/srep06663

[64] Q. Mei, J. Guo, and D. Radev, "DivRank: The interplay of prestige and diversity in information networks," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery Data Mining (KDD)*, Washington, DC, Jul. 2010, pp. 1009–1018.

[65] D. Walker, H. Xie, K.-K. Yan, and S. Maslov, "Ranking scientific publications using a model of network traffic," *J. Statist. Mechanics: Theory Experiment*, no. 06, Jun. 2007.

[66] A. P. Singh, K. Shubhankar, and V. Pudi, "An efficient algorithm for ranking research papers based on citation network," in *Proc. Conf. Data Mining Optimization (DMO)*, Bangi, Malaysia, Jun. 2011.

[67] F. Radicchi, "Who is the best player ever? a complex network analysis of the history of professional tennis," *PLOS ONE*, vol. 6, no. 2, Feb. 2011.

[68] R. A. Horn and C. R. Johnson, *Matrix Analysis.* Cambridge University Press, 2012.

[69] P. Arbenz, “Lecture notes on solving large scale eigenvalue problems,” ETH Zürich, Tech. Rep., 2012.

[70] whuber. (2019) Is PCA optimization convex? [Online]. Available: https://stats.stackexchange.com/q/301561

[71] R. Tibshirani, “Lecture notes on convex optimization,” Carnegie Mellon University, Tech. Rep., 2015.

[72] P. Xu, B. He, C. De Sa, I. Mitliagkas, and C. Re, “Accelerated stochastic power iteration,” in *Proc. Int. Conf. Artif. Intell. Statist. (AISTATS)*, Playa Blanca, Lanzarote, Canary Islands, Apr. 2018.

[73] P. Drineas, R. Kannan, and M. W. Mahoney, “Fast Monte Carlo algorithms for matrices i: Approximating matrix multiplication,” *SIAM J. Comput.*, vol. 36, no. 1, pp. 132–157, Jan. 2006.

[74] F. Held, “Theory related to PCA and SVD,” Mathematical Sciences, Chalmers University of Technology and University of Gothenburg, Tech. Rep., 2019.

[75] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Trans. ASME J. Basic Eng.*, vol. 82, no. 1, pp. 35–45, Mar. 1960.

[76] Wikipedia contributors. (2021) Kalman filter. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kalman_filter&oldid=1056993137

[77] S. Polavarapu, “The Kalman filter,” Dept. of Physics, University of Toronto, Toronto, ON, Canada, Tech. Rep., 2004. [Online]. Available: http://www.atmosp.physics.utoronto.ca/PHY2509/ch6.pdf

[78] M. Luby, “LT codes,” in *Proc. IEEE Symp. Found. Comput. Sci. (FOCS)*, Vancouver, BC, Canada, Nov. 2002, pp. 271–280.

[79] A. Shokrollahi and M. Luby, “Raptor codes,” *Found. Trends Commun. Inf. Theory*, vol. 6, no. 3–4, pp. 213–322, May 2011.

[80] M. A. Shokrollahi, S. Lassen, and R. Karp, “Systems and processes for decoding chain reaction codes through inactivation,” U.S. Patent 6,856,263, Feb., 2005.

[81] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer, “Raptor forward error correction scheme for object delivery,” Internet Requests for Comments, RFC Editor, RFC 5053, Oct. 2007.

[82] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, “RaptorQ forward error correction scheme for object delivery,” Internet Requests for Comments, RFC Editor, RFC 6330, Aug. 2011.

[83] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Barcelona, Spain, Jul. 2016, pp. 1143–1147.

[84] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, “Gradient coding: Avoiding stragglers in distributed learning,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, Sydney, Australia, Aug. 2017, pp. 3368–3376.

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

[86] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, “A unified coding framework for distributed computing with straggling servers,” in *Proc. Workshop Netw. Coding Appl. (NetCod)*, Washington, DC, 2016.

[87] A. Severinson, A. Graell i Amat, and E. Rosnes, “Block-diagonal and LT codes for distributed computing with straggling servers,” *IEEE Trans. Commun.*, vol. 67, no. 3, pp. 1739–1753, Mar. 2019.

[88] A. Severinson, A. Graell i Amat, E. Rosnes, F. Lázaro, and G. Liva, “A droplet approach based on Raptor codes for distributed computing with straggling servers,” in *Proc. Int. Symp. Turbo Codes Iterative Inf. Process. (ISTC)*, Hong Kong, China, Dec. 2018.

[89] Y. Yang, M. Chaudhari, P. Grover, and S. Kar, “Coding for a single sparse inverse problem,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, Jun. 2018, pp. 1575–1579.

[90] Q. Yu, M. Maddah-Ali, and S. Avestimehr, “Polynomial codes: an optimal design for high-dimensional coded matrix multiplication,” in *Proc. Neural Inf. Process. Syst. (NIPS)*, Long Beach, CA, Dec. 2017, pp. 4403–4413.

[91] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, “Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, Jun. 2018, pp. 2022–2026.

[92] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, “Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD,” in *Proc. Int. Conf. Artif. Intell. Statist. (AISTATS)*, Lanzarote, Canary Islands, Apr. 2018, pp. 803–812.

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

[94] M. Fahim and V. R. Cadambe, “Numerically stable polynomially coded computing,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Paris, France, Jul. 2019, pp. 3017–3021.

[95] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, “On the optimal recovery threshold of coded matrix multiplication,” *IEEE Trans. Inf. Theory*, vol. 66, no. 1, pp. 278–301, Jan. 2020.

[96] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, “OverSketch: Approximate matrix multiplication for the cloud,” in *Proc. IEEE Int. Conf. Big Data (BigData)*, Seattle, WA, Dec. 2018, pp. 298–304.

[97] T. Jahani-Nezhad and M. A. Maddah-Ali, “CodedSketch: Coded distributed computation of approximated matrix multiplication,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2019, pp. 2489–2493.

[98] C.-S. Yang, R. Pedarsani, and A. S. Avestimehr, “Timely-throughput optimal coded computing over cloud networks,” in *Proc. ACM Int. Symp. Mobile Ad Hoc Netw. Comput. (MobiHoc)*, Catania, Italy, Jul. 2019, pp. 301–310.

[99] M. Ye and E. Abbe, “Communication-computation efficient gradient coding,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, Stockholm, Sweden, Jul. 2018, pp. 5610–5619.

[100] H. Wang, Z. Charles, and D. Papailiopoulos, “ErasureHead: Distributed gradient descent without delays using approximate gradient coding,” Jan. 2019. [Online]. Available: http://arxiv.org/abs/1901.09671

[101] R. Bitar, M. Wootters, and S. El Rouayheb, “Stochastic gradient coding for straggler mitigation in distributed learning,” *IEEE J. Sel. Areas Inf. Theory*, vol. 1, no. 1, pp. 277–291, May 2020.

[102] L. Chen, H. Wang, Z. Charles, and D. Papailiopoulos, “DRACO: Byzantine-resilient distributed training via redundant gradients,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, Stockholm, Sweden, Jul. 2018, pp. 903–912.

[103] W. Halbawi, N. Azizan, F. Salehi, and B. Hassibi, “Improving distributed gradient descent using Reed-Solomon codes,” in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, Jun. 2018, pp. 2027–2031.

[104] N. Raviv, I. Tamo, R. Tandon, and A. G. Dimakis, “Gradient coding from cyclic MDS codes and expander graphs,” in *Proc. Int. Conf. Mach. Learn. (ICML)*, Stockholm, Sweden, Jul. 2018, pp. 7475–7489.

[105] H. Nagaraja, “Order statistics from independent exponential random variables and the sum of the top order statistics,” in *Advances in Distribution Theory, Order Statistics, and Inference.* Birkhäuser Boston, 2006.

[106] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[107] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, “Robust stochastic approximation approach to stochastic programming,” *SIAM J. Optimization*, vol. 19, no. 4, pp. 1574–1609, Jan. 2009.

[108] M. Schmidt, N. Le Roux, and F. Bach, “Minimizing finite sums with the stochastic average gradient,” *Math. Programming*, vol. 162, no. 1, pp. 83–112, Mar. 2017.

[109] A. Defazio, F. Bach, and S. Lacoste-Julien, "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Montreal, QC, Canada, Dec. 2014, pp. 1646–1654.

[110] C. Calauzènes and N. Le Roux, "Distributed SAGA: Maintaining linear convergence rate with limited communication," May 2017. [Online]. Available: https://arxiv.org/abs/1705.10405

[111] L. M. Nguyen, J. Liu, K. Scheinberg, and M. Takáč, "SARAH: A novel method for machine learning problems using stochastic recursive gradient," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Sydney, Australia, Aug. 2017, pp. 2613–2621.

[112] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proc. Neural Inf. Process. Syst. (NIPS)*, Lake Tahoe, NV, Dec. 2013, pp. 315–323.

[113] F. Hanzely, K. Mishchenko, and P. Richtárik, "SEGA: Variance reduction via gradient sketching," in *Proc. Neural Inf. Process. Syst. (NeurIPS)*, Montreal, QC, Canada, Dec. 2018, pp. 2082–2093.

[114] E. Gorbunov, K. Burlachenko, Z. Li, and P. Richtárik, "MARINA: Faster non-convex distributed learning with compression," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Jul. 2021, pp. 3788–3798.

Part B

Papers

Paper I

Block-diagonal and LT codes for distributed computing with straggling servers

Albin Severinson, Alexandre Graell i Amat, and Eirik Rosnes

In IEEE Transactions on Communications.

The layout has been revised.

Block-Diagonal and LT Codes for Distributed Computing With Straggling Servers

Albin Severinson*†, Alexandre Graell i Amat‡, and Eirik Rosnes*

*Simula UiB, Bergen, Norway

†Department of Informatics, University of Bergen, Bergen, Norway

‡Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden

Abstract

We propose two coded schemes for the distributed computing problem of multiplying a matrix by a set of vectors. The first scheme is based on partitioning the matrix into submatrices and applying maximum distance separable (MDS) codes to each submatrix. For this scheme, we prove that up to a given number of partitions the communication load and the computational delay (not including the encoding and decoding delay) are identical to those of the scheme recently proposed by Li *et al.*, based on a single, long MDS code. However, due to the use of shorter MDS codes, our scheme yields a significantly lower overall computational delay when the delay incurred by encoding and decoding is also considered. We further propose a second coded scheme based on Luby Transform (LT) codes under inactivation decoding. Interestingly, LT codes may reduce the delay over the partitioned scheme at the expense of an increased communication load. We also consider distributed computing under a deadline and show numerically that the proposed schemes outperform other schemes in the literature, with the LT code-based scheme yielding the best performance for the scenarios considered.

1 Introduction

Distributed computing systems have emerged as one of the most effective ways of solving increasingly complex computational problems, such as those in large-scale machine learning and data analytics [1], [2], [3]. These systems, referred to as “warehouse-scale computers” (WSCs) [1], may be composed of thousands of relatively homogeneous hardware and software components. Achieving high availability and efficiency for applications running on WSCs is a major challenge. One of the main reasons is the large number of components that may experience transient or permanent failures [3]. As a result, several distributed computing frameworks have been proposed [4], [5], [6]. In particular, MapReduce [4] 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 [4]. Among the challenges brought on by distributed computing systems, 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, which may cause large delays [4]. 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.

Coding for distributed computing to reduce the computational delay and the communication load between servers has recently been considered in [7], [8]. 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 maximum distance separable (MDS) 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. 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 and the computational delay, respectively. The code proposed in [9] is an MDS code of code length proportional to the number of rows of the matrix to be multiplied, which may be very large in practice. 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 [10]. In [7], [8], [9], the computational delay incurred by the encoding and decoding is not considered. However, the encoding and decoding may incur a high computational delay for large matrices.

Coding has previously been applied to several related problems in distributed computing. For example, the scheme in [8] has been extended to distributed matrix-matrix multiplication where both matrices are

too large to be stored at one server [11], [12]. Whereas the schemes in [8], [11] are based on MDS codes, the scheme in [12] is based on a novel coding scheme that exploits the algebraic properties of matrix-matrix multiplication over a finite field to reduce the computational delay. In [13], it was shown that introducing sparsity in a structured manner during encoding can speed up computing dot products between long vectors. Distributed computing over heterogeneous clusters has been considered in [14].

In this paper, we propose two coding schemes for the problem of multiplying a matrix by a set of vectors. The first is a block-diagonal coding (BDC) scheme equivalent to partitioning the matrix and applying smaller MDS codes to each submatrix separately (we originally introduced the BDC scheme in [15]). The storage design for the BDC scheme can be cast as an integer optimization problem, 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. Furthermore, we prove that up to a certain level of partitioning the BDC scheme has identical computational delay (as defined in [9]) and communication load to those of the scheme in [9]. Interestingly, when the delay incurred by encoding and decoding is taken into account, the proposed scheme achieves an overall computational delay significantly lower than that of the scheme in [9]. We further propose a second coding scheme based on Luby Transform (LT) codes [16] under inactivation decoding [17], which in some scenarios achieves a lower computational delay than that of the BDC scheme at the expense of a higher communication load. We show that for the LT code-based scheme it is possible to trade an increase in communication load for a lower computational delay. We finally consider distributed computing under a deadline, where we are interested in completing a computation within some computational delay, and show numerically that both the BDC and the LT code-based schemes significantly increase the probability of meeting a deadline over the scheme in [9]. In particular, the LT code-based scheme achieves the highest probability of meeting a deadline for the scenarios considered.

2 System Model and Preliminaries

We consider the distributed matrix multiplication problem, i.e., the problem of multiplying a set of vectors with a matrix. In particular, given an $m \times n$ matrix $\boldsymbol{A} \in \mathbb{F}_{2^l}^{m\times n}$ and N vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N \in \mathbb{F}_{2^l}^{n}$, where $\mathbb{F}_{2^l}$ is an extension field of characteristic 2, we want to compute the N vectors $\boldsymbol{y}_1 = \boldsymbol{A}\boldsymbol{x}_1, \dots, \boldsymbol{y}_N = \boldsymbol{A}\boldsymbol{x}_N$. The computation is performed in a distributed fashion using K servers, $S_1, \dots, S_K$. Each server is responsible for multiplying ηm matrix rows by the vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$, for some $\frac{1}{K} \leq \eta \leq 1$. We refer to η as the fraction of rows stored at each server and we assume that η is selected such that ηm is an integer. Prior to computing $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$, $\boldsymbol{A}$ is encoded by an $r \times m$ encoding matrix $\boldsymbol{\Psi} = [\Psi_{i,j}]$, 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 carried out in a distributed manner over the K servers and is used to alleviate the straggler problem. We allow assigning each row of the coded matrix $\boldsymbol{C}$ to several servers to enable coded multicasting, a strategy used to address the bandwidth scarcity problem. Let

$$q = K\frac{m}{r},$$

where we assume that r divides Km and hence q is an integer. The r coded rows of $\boldsymbol{C}$, $\boldsymbol{c}_1, \dots, \boldsymbol{c}_r$, are divided into $\binom{K}{\eta q}$ disjoint batches, each containing $r/\binom{K}{\eta q}$ coded rows. Each batch is assigned to ηq servers. Correspondingly, a batch B is labeled by a unique set $\mathcal{T} \subset \{S_1, \dots, S_K\}$, of size $|\mathcal{T}| = \eta q$, 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, $k = 1, \dots, K$, stores the coded rows of $B_{\mathcal{T}}$ if and only if $S_k \in \mathcal{T}$.

2.1 Probabilistic Runtime Model

We assume that running a computation on a single server takes a random amount of time, which is denoted by the random variable H, according to the shifted-exponential cumulative probability distribution function (CDF)

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

where σ is a parameter used to scale the distribution. Denote by σ_{A} and σ_{M} the number of time units required to complete one addition and one multiplication (over $\mathbb{F}_{2^l}$), respectively, over a single server. Let σ be the weighted sum of the number of additions and multiplications required to complete the computation, where the weighting coefficients are σ_{A} and σ_{M}, respectively. As in [18], we assume that σ_{A} is in $\mathcal{O}(\frac{l}{64})$ and σ_{M} in $\mathcal{O}(l \log_2 l)$. Furthermore, we assume that the hidden coefficients are comparable and will thus not consider them. With some abuse of language, we refer to the parameter σ associated with some computation as its computational complexity. For example, the complexity (number of time units) of computing the inner product of two length-n vectors is $\sigma = (n-1)\sigma_{\mathsf{A}} + n\sigma_{\mathsf{M}}$ as it requires performing $n - 1$ additions and n multiplications. The shift of the shifted-exponential distribution should be interpreted as the minimum amount of time the computation can be completed in. The tail of the distribution accounts for transient disturbances that are at the root of the straggler problem. These include transmission and queuing delays

during initialization as well as contention for the local disk and slow-downs due to higher priority tasks being assigned to the same server [19]. The complexity of a computation σ affects both the shift and the tail of the distribution since the probability of transient behavior occurring increases with the amount of time the computation is running. In the results section we also consider a model where σ only affects the shift. The shifted-exponential distribution was proposed as a model for the latency of file queries from cloud storage systems in [20] and was subsequently used to model computational delay in [8], [9].

When an 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 H_k. As in [8], we assume that $H_1, \dots, H_K$ are independent and identically distributed random variables with CDF $F_H(Kh;\sigma)$. For $i = 1, \dots, K$, we denote the i-th order statistic by $H_{(i)}$, i.e., the i-th smallest random variable of $H_1, \dots, H_K$. The runtime of the i-th fastest server to complete its subtask is thus given by $H_{(i)}$, which is a Gamma distributed random variable with expectation and variance given by [21]

$$\mu(\sigma, K, g) \triangleq \mathbb{E}\left(H_{(i)}\right) = \sigma\left(1 + \sum_{j=K-i+1}^{K} \frac{1}{j}\right), \ \mathrm{Var}\left(H_{(i)}\right) = \sigma^2 \sum_{j=K-i+1}^{K} \frac{1}{j^2}.$$

We parameterize the Gamma distribution by its inverse scale factor a and its shape parameter b. We give these in terms of the distribution mean and variance as [22]

$$a = \frac{\mathbb{E}[H_{(i)}] - \sigma}{\mathrm{Var}[H_{(i)}]} \ \text{ and } \ b = \frac{\left(\mathbb{E}[H_{(i)}] - \sigma\right)^2}{\mathrm{Var}[H_{(i)}]}.$$

Denote by $F_{H_{(i)}}(h_{(i)};\sigma, K)$ the CDF of $H_{(i)}$. It is given by [22]

$$F_{H_{(i)}}(h_{(i)};\sigma, K) = \begin{cases} \frac{\gamma(b, a(h_{(i)}-\sigma))}{\Gamma(b)}, & \text{for } h_{(i)} \geq \sigma \\ 0, & \text{otherwise} \end{cases},$$

where Γ denotes the Gamma function and γ the lower incomplete Gamma function,

$$\Gamma(b) = \int_0^\infty x^{b-1} \mathrm{e}^{-x}\, \mathrm{d}x \ \text{ and } \ \gamma(b, ah) = \int_0^{ah} x^{b-1} \mathrm{e}^{-x}\, \mathrm{d}x.$$

We remark that $F_{H_{(i)}}(h_{(i)};\sigma, K)$ is the probability of a computation finishing prior to some deadline $t = h_{(i)}$.

2.2 Distributed Computing Model

We consider the coded computing framework introduced in [9], which extends the MapReduce framework [4]. The overall computation proceeds in three phases, the *map*, *shuffle*, and *reduce* phases, which are augmented to make use of the coded multicasting strategy proposed in [7] to address the bandwidth scarcity problem and the coded scheme proposed in [8] to alleviate the straggler problem. Furthermore, we consider the delay incurred by the encoding of $\boldsymbol{A}$ that takes place before the start of the map phase. We refer to this as the *encoding* phase. Also, we assume that the matrices $\boldsymbol{A}$ and $\boldsymbol{\Psi}$ as well as the input vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$ are known to all servers at the start of the computation. The overall computation proceeds in the following manner.

2.2.1 Encoding Phase

In the encoding phase, the coded matrix $\boldsymbol{C}$ is computed from $\boldsymbol{A}$ and $\boldsymbol{\Psi}$ in a distributed fashion. Specifically, denote by $\mathcal{R}^{(S)}$ the set of indices of rows of $\boldsymbol{C}$ that are assigned to server S and denote by $\boldsymbol{\Psi}^{(S)}$ the matrix consisting of the rows of $\boldsymbol{\Psi}$ with indices from $\mathcal{R}^{(S)}$. Then, server S computes the coded rows it needs by multiplying $\boldsymbol{\Psi}^{(S)}$ by $\boldsymbol{A}$. Note that since we assign each coded row to ηq servers, each row of $\boldsymbol{C}$ is computed separately by ηq servers. We define the computational delay of the encoding phase as its average runtime per source row and vector $\boldsymbol{y}$, i.e.,

$$D_{\mathsf{encode}} = \frac{\eta q}{mN} \mu\left(\frac{\sigma_{\mathsf{encode}}}{K}, K, K\right),$$

where σ_{encode} is the complexity of the encoding. During the encoding process, the rows of $\boldsymbol{\Psi}$ are multiplied by the columns of $\boldsymbol{A}$. Therefore, the complexity scales with the product of the number of nonzero elements of $\boldsymbol{\Psi}$ and the number of columns of $\boldsymbol{A}$. Specifically,

$$\sigma_{\mathsf{encode}} = \left|\{(i,j) : \Psi_{i,j} \neq 0\}\right| n\left(\sigma_{\mathsf{A}} + \sigma_{\mathsf{M}}\right) - n\sigma_{\mathsf{A}}.$$

Alternatively, we compute $\boldsymbol{C}$ by performing a decoding operation on $\boldsymbol{A}$. In this case σ_{encode} is the decoding complexity (see Section 4.2). Furthermore, since the decoding algorithms are designed to decode the entire codeword, each server has to compute all rows of $\boldsymbol{C}$. Using this strategy the encoding delay is

$$D_{\mathsf{encode}} = \frac{K}{mN} \mu\left(\frac{\sigma_{\mathsf{encode}}}{K}, K, K\right).$$

For each case we choose the strategy that minimizes the delay.

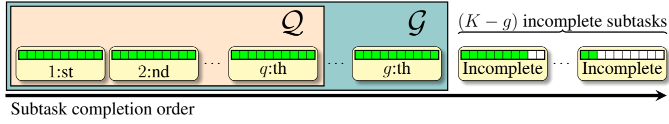


Fig. 1: Servers (yellow boxes) finish their respective subtasks in random order.

2.2.2 Map Phase

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

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

The map phase terminates when a set of servers $\mathcal{G} \subseteq \{S_1, \dots, S_K\}$ 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 $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ can be obtained from any subset of q servers, i.e., $g = q$. We illustrate the completion of subtasks in Fig. 1.

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

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

where $\sigma_{\mathsf{map}} = K\eta mN\left((n-1)\sigma_{\mathsf{A}} + n\sigma_{\mathsf{M}}\right)$, as all K servers compute ηm inner products, each requiring $n-1$ additions and n multiplications, for each of the N input vectors. In [9], D_{map} is referred to simply as the computational delay.

After the map phase, the computation of $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ 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 $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$. Let $\mathcal{W}_S$ be the set containing the indices of the vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ that server $S \in \mathcal{Q}$ is responsible for. The remaining servers in $\mathcal{G}$ assist the servers in $\mathcal{Q}$ in the shuffle phase.

2.2.3 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. Furthermore, as in [8], we denote by $\phi(j)$ the ratio between the communication load of unicasting the same message to each of j recipients and multicasting that message to j recipients. For example, if the communication load of multicasting a message to j recipients and unicasting a message to a single recipient is the same, we have $\phi(j) = j$. On the other hand, if the communication load of multicasting a message to j recipients is equal to that of unicasting that same message to each recipient, $\phi(j) = 1$. The total communication load of a multicast message is then given by $\frac{j}{\phi(j)}$. The shuffle phase proceeds in three steps as follows.

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 $\mathcal{Q}$, and each coded message is simultaneously sent to multiple servers. We denote by

$$s_q \triangleq \inf\left(s : \sum_{j=s}^{\eta q} \alpha_j \le 1 - \eta\right), \; \alpha_j \triangleq \frac{\binom{q-1}{j}\binom{K-q}{\eta q - j}}{\frac{q}{K}\binom{K}{\eta q}}, \tag{1}$$

the smallest number of recipients of a coded message [9]. We remark that $m\alpha_j$ is the total number of coded values delivered to each server via the coded multicast messages with exactly j recipients. More specifically, for each $j \in \{\eta q, \eta q - 1, \dots, s_q\}$, 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 $\mathcal{V}_{\mathcal{S}\setminus S}^{(S)}$ into j disjoint segments, $\mathcal{V}_{\mathcal{S}\setminus S}^{(S)} = \{\mathcal{V}_{\mathcal{S}\setminus S, \tilde{S}}^{(S)} : \tilde{S} \in \mathcal{S} \setminus S\}$, and associate the segment $\mathcal{V}_{\mathcal{S}\setminus S, \tilde{S}}^{(S)}$ to server $\tilde{S}$.
2) Server $\tilde{S} \in \mathcal{S}$ multicasts the bit-wise modulo-2 sum of all the segments associated to it in $\mathcal{S}$. More precisely, it multicasts $\oplus_{S \in \mathcal{S}\setminus\tilde{S}} \mathcal{V}_{\mathcal{S}\setminus S, \tilde{S}}^{(S)}$ to the other servers in $\mathcal{S} \setminus \tilde{S}$, where $\oplus$ denotes the modulo-2 sum operator.

By construction, exactly one value that each coded message is composed of is unknown to each recipient. The other values have been computed locally by the recipient. 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 $j = s_q - 1$. We refer to these alternatives as shuffling strategy 1 and 2, respectively. We always select the strategy achieving the lowest communication load. If any server in $\mathcal{Q}$ still needs more intermediate values at this point, they are unicasted from other servers in $\mathcal{G}$. This may happen only if a non-MDS code is used. We remark that it may be possible to opportunistically create additional coded multicasting opportunities by exploiting the remaining $g - q$ servers in $\mathcal{G}$.

Definition 1. *The communication load, denoted by L, is the number of unicasts and multicasts (weighted by their cost relative to a unicast) per source row and vector $\boldsymbol{y}$ exchanged during the shuffle phase. Specifically, each unicasted message increases L by $\frac{1}{mN}$, and each message multicasted to j recipients increases L by $\frac{j}{mN\phi(j)}$.*

The communication load after completing the shuffle phase is given in [9]. If the shuffle phase finishes by unicasting the remaining needed values (strategy 1), the communication load after completing the multicast phase is

$$\sum_{j=s_q}^{\eta q} \frac{\alpha_j}{\phi(j)}.$$

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

$$\sum_{j=s_q-1}^{\eta q} \frac{\alpha_j}{\phi(j)}.$$

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

$$L_{\mathsf{MDS}} = \min\left(\sum_{j=s_q}^{\eta q} \frac{\alpha_j}{\phi(j)} + 1 - \eta - \sum_{j=s_q}^{\eta q} \alpha_j, \sum_{j=s_q-1}^{\eta q} \frac{\alpha_j}{\phi(j)}\right), \tag{2}$$

where $1 - \eta - \sum_{j=s_q}^{\eta q} \alpha_j$ is the communication load due to unicasting the remaining needed values.

2.2.4 Reduce Phase

Finally, in the reduce phase, the vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N$ are computed. More specifically, server $S \in \mathcal{Q}$ uses the locally computed sets $\mathcal{Z}_1^{(S)}, \ldots, \mathcal{Z}_N^{(S)}$ 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.,

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

where σ_{reduce} is the computational complexity (see Section 2.1) of the reduce phase.

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

2.3 Previously Proposed Coded Computing Schemes

Here we formally define the uncoded scheme (UC) and the coded computing schemes of [7], [8], [9] (which we refer to as the straggler coding (SC), coded MapReduce (CMR), and unified scheme, respectively) in terms of the model above. Specifically, to make a fair comparison with our coded computing scheme with parameters K, q, m, and η, we define the corresponding uncoded, CMR, SC, and unified schemes. When referring to the system parameters of a given scheme, we will write the scheme acronym in the subscript. We only explicitly mention the parameters that differ. The number of servers K is unchanged for all schemes considered.

The uncoded scheme uses no erasure coding and no coded multicasting and has parameters $\eta_{\mathsf{UC}} = \frac{1}{K}$ and $q_{\mathsf{UC}} = K$, implying $\eta_{\mathsf{UC}} q_{\mathsf{UC}} = 1$. Furthermore, the encoding matrix $\boldsymbol{\Psi}_{\mathsf{UC}}$ is the $m \times m$ identity matrix and the coded matrix is $\boldsymbol{C}_{\mathsf{UC}} = \boldsymbol{A}$.

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

The SC scheme [8] uses an erasure code but no coded multicasting. For the corresponding SC scheme, the code rate is unchanged, i.e., $q_{\mathsf{SC}} = q$, and the fraction of rows stored at each server is $\eta_{\mathsf{SC}} = \frac{1}{q_{\mathsf{SC}}}$. The encoding matrix $\boldsymbol{\Psi}_{\mathsf{SC}}$ of the SC scheme is obtained by splitting the rows of $\boldsymbol{A}$ into q_{SC} equally tall submatrices

$$\boldsymbol{C} = \boldsymbol{\Psi}_{\mathsf{BDC}}\boldsymbol{A} = \begin{bmatrix} \boldsymbol{\psi}_1 & & \\ & \boldsymbol{\psi}_2 & \\ & & \boldsymbol{\psi}_3 \end{bmatrix} \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \boldsymbol{A}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\psi}_1\boldsymbol{A}_1 \\ \boldsymbol{\psi}_2\boldsymbol{A}_2 \\ \boldsymbol{\psi}_3\boldsymbol{A}_3 \end{bmatrix}$$

Fig. 2: BDC scheme with $T = 3$ partitions.

$\boldsymbol{A}_1, \dots, \boldsymbol{A}_{q_{\mathsf{SC}}}$ 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 $\boldsymbol{C}_{\mathsf{SC}}$ is the concatenation of $\boldsymbol{C}_1, \dots, \boldsymbol{C}_K$, i.e.,

$$\boldsymbol{C}_{\mathsf{SC}} = \begin{pmatrix} \boldsymbol{C}_1 \\ \vdots \\ \boldsymbol{C}_K \end{pmatrix}.$$

The unified scheme [9] uses both an erasure code and coded multicasting and has parameters $\eta_{\mathsf{unified}} = \eta$ and $q_{\mathsf{unified}} = q$. Furthermore, the encoding matrix of the unified scheme, $\boldsymbol{\Psi}_{\mathsf{unified}}$, is an (r, m) MDS code encoding matrix.

3 Block-Diagonal Coding

In this section, we introduce a BDC scheme for the problem of multiplying a matrix by a set of vectors. For large matrices, the encoding and decoding complexity of the proposed scheme is significantly lower than that of the scheme in [9], leading to a lower overall computational delay, as will be shown in Section 7. Specifically, the scheme is based on a block-diagonal encoding matrix of the form

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

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 $\boldsymbol{\Psi}_{\mathsf{BDC}}$ 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 $\boldsymbol{\Psi}_{\mathsf{BDC}}$ with T disjoint submatrices as a T-partitioned scheme, and to the submatrix of $\boldsymbol{C} = \boldsymbol{\Psi}_{\mathsf{BDC}}\boldsymbol{A}$ 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. Notably, by keeping the ratio $\frac{m}{T}$ constant, the decoding complexity scales linearly with m. We further remark that the case $\boldsymbol{\Psi}_{\mathsf{BDC}} = \boldsymbol{\psi}$ (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 BDC scheme with $T = 3$ partitions in Fig. 2.

3.1 Assignment of Coded Rows to Batches

For a block-diagonal encoding matrix $\boldsymbol{\Psi}_{\mathsf{BDC}}$, 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 Section 2, the coded rows are divided into $\binom{K}{\eta q}$ disjoint batches. To formally describe the assignment of coded rows to batches we use a $\binom{K}{\eta q} \times T$ integer matrix $\boldsymbol{P} = [p_{i,j}]$, where $p_{i,j}$ is the number of rows from partition j that are stored in batch i. In the sequel, $\boldsymbol{P}$ will be referred to as the assignment matrix. 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. The rows of $\boldsymbol{P}$ are labeled by the subset of servers the corresponding batch is stored at, and the columns are labeled by their partition indices. For convenience, we refer to the pair $(\boldsymbol{\Psi}_{\mathsf{BDC}}, \boldsymbol{P})$ 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 up to the batch size, i.e.,

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

$\boldsymbol{c}_1^{(1)}\ \boldsymbol{c}_3^{(1)}\ \boldsymbol{c}_5^{(1)}\ \boldsymbol{c}_1^{(2)}\ \boldsymbol{c}_3^{(2)}$ $\boldsymbol{c}_2^{(1)}\ \boldsymbol{c}_4^{(1)}\ \boldsymbol{c}_6^{(1)}\ \boldsymbol{c}_2^{(2)}\ \boldsymbol{c}_4^{(2)}$ **Server $\boldsymbol{S}_1$**	$\boldsymbol{c}_1^{(1)}\ \boldsymbol{c}_5^{(2)}\ \boldsymbol{c}_1^{(3)}\ \boldsymbol{c}_3^{(3)}\ \boldsymbol{c}_5^{(3)}$ $\boldsymbol{c}_2^{(1)}\ \boldsymbol{c}_6^{(2)}\ \boldsymbol{c}_2^{(3)}\ \boldsymbol{c}_4^{(3)}\ \boldsymbol{c}_6^{(3)}$ **Server $\boldsymbol{S}_2$**	$\boldsymbol{c}_3^{(1)}\ \boldsymbol{c}_5^{(2)}\ \boldsymbol{c}_1^{(4)}\ \boldsymbol{c}_3^{(4)}\ \boldsymbol{c}_5^{(4)}$ $\boldsymbol{c}_4^{(1)}\ \boldsymbol{c}_6^{(2)}\ \boldsymbol{c}_2^{(4)}\ \boldsymbol{c}_4^{(4)}\ \boldsymbol{c}_6^{(4)}$ **Server $\boldsymbol{S}_3$**
$\boldsymbol{c}_5^{(1)}\ \boldsymbol{c}_1^{(3)}\ \boldsymbol{c}_1^{(4)}\ \boldsymbol{c}_1^{(5)}\ \boldsymbol{c}_3^{(5)}$ $\boldsymbol{c}_6^{(1)}\ \boldsymbol{c}_2^{(3)}\ \boldsymbol{c}_2^{(4)}\ \boldsymbol{c}_2^{(5)}\ \boldsymbol{c}_4^{(5)}$ **Server $\boldsymbol{S}_4$**	$\boldsymbol{c}_1^{(2)}\ \boldsymbol{c}_3^{(3)}\ \boldsymbol{c}_3^{(4)}\ \boldsymbol{c}_1^{(5)}\ \boldsymbol{c}_5^{(5)}$ $\boldsymbol{c}_2^{(2)}\ \boldsymbol{c}_4^{(3)}\ \boldsymbol{c}_4^{(4)}\ \boldsymbol{c}_2^{(5)}\ \boldsymbol{c}_6^{(5)}$ **Server $\boldsymbol{S}_5$**	$\boldsymbol{c}_3^{(2)}\ \boldsymbol{c}_5^{(3)}\ \boldsymbol{c}_5^{(4)}\ \boldsymbol{c}_3^{(5)}\ \boldsymbol{c}_5^{(5)}$ $\boldsymbol{c}_4^{(2)}\ \boldsymbol{c}_6^{(3)}\ \boldsymbol{c}_6^{(4)}\ \boldsymbol{c}_4^{(5)}\ \boldsymbol{c}_6^{(5)}$ **Server $\boldsymbol{S}_6$**

Fig. 3: Storage design for $m = 20$, $N = 4$, $K = 6$, $q = 4$, $\eta = 1/2$, and $T = 5$.

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

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

We clarify the assignment of coded rows to batches and the coded computing scheme in the following example.

Example 1 ($m = 20$, $N = 4$, $K = 6$, $q = 4$, $\eta = 1/2$, $T = 5$)**.** *For these parameters, there are $r/T = 6$ coded rows per partition, of which $m/T = 4$ are sufficient for decoding, and $\binom{K}{\eta q} = 15$ batches, each containing $r/\binom{K}{\eta q} = 2$ coded rows. We construct the storage design shown in Fig. 3 with $\binom{K}{\eta q} \times T = 15 \times 5$ assignment matrix*

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

where rows are labeled by the subset of servers the batch is stored at, and columns are labeled by the partition index. In this case 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. 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 that $\mathcal{G} = \{S_1, S_2, S_3, S_4\}$ 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. Server 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 (2)*). In this case, S_1 received additional intermediate values from partition 2, despite already storing enough, further indicating that the assignment in* (3) *is suboptimal.*

4 Performance of the Block-Diagonal Coding

In this section, we analyze the impact of partitioning on the performance. We also prove that we can partition up to the batch size, i.e., $T = r/\binom{K}{\eta q}$, without increasing the communication load and the computational delay of the map phase with respect to the original scheme in [9].

4.1 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 BDC (partitioned) scheme, both g and the number of remaining unicasts may vary.

For a given assignment matrix $\boldsymbol{P}$ and a specific $\mathcal{Q}$, we denote by $U_{\mathcal{Q}}^{(S)}(\boldsymbol{P})$ the number of remaining values needed after the multicast phase by server $S \in \mathcal{Q}$, and by

$$U_{\mathcal{Q}}(\boldsymbol{P}) \triangleq \sum_{S \in \mathcal{Q}} U_{\mathcal{Q}}^{(S)}(\boldsymbol{P}) \tag{4}$$

the total number of remaining values needed by the servers in $\mathcal{Q}$. Note that both $U_{\mathcal{Q}}^{(S)}(\boldsymbol{P})$ and $U_{\mathcal{Q}}(\boldsymbol{P})$ depend on the strategy used to finish the shuffle phase (see Section 2.2.3). We remark that all sets $\mathcal{Q}$ are equally likely.

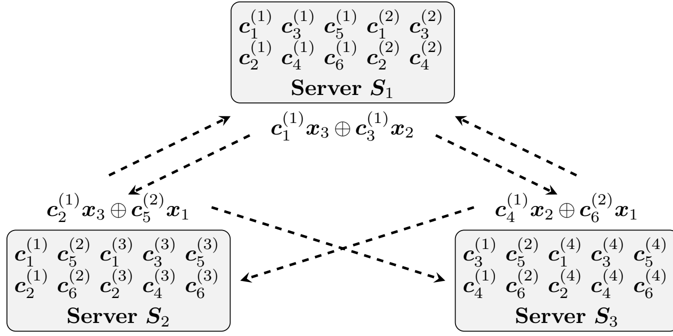


Fig. 4: Multicasting coded values between servers S_1, S_2, and S_3.

Let $\mathbb{Q}^q$ denote the superset of all sets $\mathcal{Q}$. Furthermore, we denote by $L_{\mathbb{Q}}(\boldsymbol{P})$ the average communication load of the messages that are unicasted after the multicasting step (see Section 2.2.3), i.e.,

$$L_{\mathbb{Q}}(\boldsymbol{P}) \triangleq \frac{1}{mN}\frac{1}{|\mathbb{Q}^q|}\sum_{\mathcal{Q}\in\mathbb{Q}^q} U_{\mathcal{Q}}(\boldsymbol{P}). \tag{5}$$

When needed we write $L_{\mathbb{Q}}^{(1)}(\boldsymbol{P})$ and $L_{\mathbb{Q}}^{(2)}(\boldsymbol{P})$, where the superscript denotes the strategy used to finish the shuffle phase. For a given storage design $(\boldsymbol{\Psi}_{\text{BDC}}, \boldsymbol{P})$, the communication load of the BDC scheme is given by

$$L_{\text{BDC}}(\boldsymbol{\Psi}_{\text{BDC}}, \boldsymbol{P}) = \min\left(\sum_{j=s_q}^{\eta q}\frac{\alpha_j}{\phi(j)} + L_{\mathbb{Q}}^{(1)}(\boldsymbol{P}), \sum_{j=s_q-1}^{\eta q}\frac{\alpha_j}{\phi(j)} + L_{\mathbb{Q}}^{(2)}(\boldsymbol{P})\right). \tag{6}$$

Note that the load due to the multicast phase is independent of the level of partitioning. Furthermore, for the unpartitioned scheme $L_{\mathbb{Q}}^{(2)} = 0$ by design.

We first explain how $U_{\mathcal{Q}}^{(S)}$ is evaluated. Let $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ be a vector of length T, where the t-th element is the number of intermediate values from partition t stored by server S at the end of the multicast phase. Furthermore, each row of $\boldsymbol{P}$ corresponds to a batch, and coded multicasting is made possible by storing each batch at multiple servers. The intermediate values transmitted during the multicast phase thus correspond to rows of $\boldsymbol{P}$. The vector $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ is then computed by adding some set of rows of $\boldsymbol{P}$. The indices of the rows to add depend on $\mathcal{Q}$ and S (see Section 2.2.3).

We denote by $\left(\boldsymbol{u}_{\mathcal{Q}}^{(S)}\right)_t$ the t-th element of the vector $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$. The number of values $U_{\mathcal{Q}}^{(S)}$ is given by adding the number of intermediate values still needed for each partition, i.e.,

$$U_{\mathcal{Q}}^{(S)} = \sum_{t=1}^{T}\max\left(\frac{m}{T} - \left(\boldsymbol{u}_{\mathcal{Q}}^{(S)}\right)_t, 0\right). \tag{7}$$

Its sum over all $S \in \mathcal{Q}$ gives $U_{\mathcal{Q}}(\boldsymbol{P})$ (see (4)). Averaging $U_{\mathcal{Q}}(\boldsymbol{P})$ over all $\mathcal{Q}$ and normalizing yields $L_{\mathbb{Q}}(\boldsymbol{P})$ (see (5)).

Example 2 (Computing $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$). *We consider the same system as in Example 1. We again assume that $\mathcal{G} = \mathcal{Q} = \{S_1, S_2, S_3, S_4\}$ is the set of $g = q = 4$ servers that finish their map computations first. During the multicast phase server S_1 receives the intermediate values in $\mathcal{V}_{\mathcal{S}\setminus S_1}^{(S_1)}$ for all sets $\mathcal{S}$ of cardinality $j + 1 = 3$ (see Section 2.2.3). In this case, we perform coded multicasting within the sets*

- $\mathcal{S} = \{S_1, S_2, S_3\}$, $\mathcal{V}_{\mathcal{S}\setminus S_1}^{(S_1)} = \{\boldsymbol{c}_5^{(2)}\boldsymbol{x}_1, \boldsymbol{c}_6^{(2)}\boldsymbol{x}_1\}$,
- $\mathcal{S} = \{S_1, S_2, S_4\}$, $\mathcal{V}_{\mathcal{S}\setminus S_1}^{(S_1)} = \{\boldsymbol{c}_1^{(3)}\boldsymbol{x}_1, \boldsymbol{c}_2^{(3)}\boldsymbol{x}_1\}$,
- $\mathcal{S} = \{S_1, S_3, S_4\}$, $\mathcal{V}_{\mathcal{S}\setminus S_1}^{(S_1)} = \{\boldsymbol{c}_1^{(4)}\boldsymbol{x}_1, \boldsymbol{c}_2^{(4)}\boldsymbol{x}_1\}$.

Note that $\mathcal{V}_{\{S_2,S_3\}}^{(S_1)}$ contains the intermediate values computed from the coded rows stored in the batch that labels the 6-th row of the assignment matrix $\boldsymbol{P}$. In the same manner, $\mathcal{V}_{\{S_2,S_4\}}^{(S_1)}$ and $\mathcal{V}_{\{S_3,S_4\}}^{(S_1)}$ correspond to rows 7 and 10 of $\boldsymbol{P}$, respectively. Furthermore, prior to the shuffle phase server S_1 stores the batches corresponding to rows 1 to 5 of $\boldsymbol{P}$. Thus, $\boldsymbol{u}_{\{S_1,S_2,S_3,S_4\}}^{(S_1)}$ is equal to the sum of rows $1, 2, 3, 4, 5, 6, 7$, and 10 of $\boldsymbol{P}$. In this case, $\boldsymbol{u}_{\{S_1,S_2,S_3,S_4\}}^{(S_1)} = (6, 6, 2, 2, 0)$, and S_1 needs 8 more intermediate values, i.e., $U_{\{S_1,S_2,S_3,S_4\}}^{(S_1)} = 8$. Computing $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ for arbitrary $\mathcal{Q}$ and S then corresponds to summing the rows of $\boldsymbol{P}$ corresponding to batches either stored by server S prior to the shuffle phase or received by S in the multicast phase. The row indices are computed as explained in Section 2.2.3.

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

$$\min_{\boldsymbol{P}\in\mathbb{P}} L_{\mathrm{BDC}}(\mathbf{\Psi}_{\mathsf{BDC}}, \boldsymbol{P}),$$

where $\mathbb{P}$ is the set of all assignments $\boldsymbol{P}$. 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 (there are $q\binom{K}{q}$ vectors $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$). We address the optimization of the assignment matrix $\boldsymbol{P}$ in Section 5.

4.2 Computational Delay

We consider the delay incurred by the encoding, map, and reduce phases (see Definition 2). As in [9], 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 encoding and reduce phases. Thus, we consider the overall computational delay

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

The encoding delay D_{encode} is a function of the number of nonzero elements of $\mathbf{\Psi}_{\mathsf{BDC}}$. As there are at most $\frac{m}{T}$ nonzero elements in each row of a block-diagonal encoding matrix, for an encoding scheme with T partitions we have

$$\sigma_{\mathsf{encode,BDC}} \leq \frac{m}{T} rn\sigma_{\mathsf{M}} + \left(\frac{m}{T} - 1\right) rn\sigma_{\mathsf{A}}. \tag{8}$$

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 a Reed-Solomon (RS) code and is decoded using either the Berlekamp-Massey (BM) algorithm or the FFT-based algorithm proposed in [23], whichever yields the lowest complexity. To the best of our knowledge the algorithm proposed in [23] is the lowest complexity algorithm for decoding long RS codes. We measure the decoding complexity by its associated shifted-exponential parameter σ (see Section 2.1).

The number of field additions and multiplications required to decode an $(r/T, m/T)$ RS code using the BM algorithm is $(r/T)\,(\xi(r/T) - 1)$ and $(r/T)^2\xi$, respectively, where ξ is the fraction of erased symbols [24]. With ξ upper bounded 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 upper bounded as

$$\sigma_{\mathsf{reduce,BDC}}^{\mathsf{BM}} \leq N\left(\sigma_{\mathsf{A}}\left(\frac{r^2(1-\frac{q}{K})}{T} - r\right) + \sigma_{\mathsf{M}}\frac{r^2(1-\frac{q}{K})}{T}\right). \tag{9}$$

On the other hand, the FFT-based algorithm has complexity $\mathcal{O}(r\log r)$ [23]. We estimate the number of additions and multiplications required for a given code length r by fitting a curve of the form $a + br\log_2(cr)$, where (a, b, c) are coefficients, to empiric results derived from the authors' implementation of the algorithm. For additions the resulting parameters are $(2, 8.5, 0.867)$ and for multiplications they are $(2, 1, 4)$. The resulting curves diverge negligibly at the measured points. The total decoding complexity for the FFT-based algorithm is

$$\sigma_{\mathsf{reduce,BDC}}^{\mathsf{FFT}} = NT\sigma_{\mathsf{A}}\left(2 + \frac{8.5r}{T}\log_2\left(0.867r/T\right)\right) + NT\sigma_{\mathsf{M}}\left(2 + \frac{r}{T}\log_2\left(4r/T\right)\right). \tag{10}$$

The encoding and decoding complexity of the unified scheme in [9] is given by evaluating (8) and either (9) or (10) (whichever gives the lowest complexity), respectively, for $T = 1$. For the BDC scheme, by choosing T close to r we can thus significantly lower the delay of the encoding and reduce phases. On the other hand, the scheme in [8] uses codes of length proportional to the number of servers K. The encoding and decoding complexity of the SC scheme in [8] is thus given by evaluating (8) and either (9) or (10) for $T = \frac{m}{q}$.

4.3 Lossless Partitioning

Theorem 1. *For $T \leq r/\binom{K}{\eta q}$, there exists an assignment matrix $\boldsymbol{P}$ such that the communication load and the 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 $T = r/\binom{K}{\eta q}$ we let $\boldsymbol{P}$ be a $\binom{K}{\eta q} \times T$ 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 $\frac{\eta q m}{T}$ rows from each partition, and since each coded row is stored by at most ηq servers, any q servers collectively store at least $\frac{\eta q m}{\eta q T} = \frac{m}{T}$ unique coded rows from each partition. The computational delay of the map phase is thus unchanged from the unpartitioned scheme. The communication load is unchanged if $U_{\mathcal{Q}}^{(S)}$ is equal to that of the unpartitioned scheme for all $\mathcal{Q}$ and S. The

number of values needed $U_{\mathcal{Q}}^{(S)}$ is computed from $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ (see (7)), which is the sum of l rows of $\boldsymbol{P}$, for some integer l. For the all-ones assignment matrix, because all rows of $\boldsymbol{P}$ are identical, we have

$$U_{\mathcal{Q}}^{(S)} = T \max\left(\frac{m}{T} - l, 0\right) = \max\left(m - Tl, 0\right),$$

which is the number of remaining values for the unpartitioned scheme.

Next, we consider the case where $T < r/\binom{K}{\eta q}$. First, consider the case $T = r/\binom{K}{\eta q} - j$, for some integer j, $0 \leq j < \frac{r}{2\binom{K}{\eta q}}$. We first set all entries of $\boldsymbol{P}$ equal to 1. At this point, the total number of unique rows of $\boldsymbol{C}$ per partition stored by any set of q servers is at least

$$\frac{m}{r/\binom{K}{\eta q}} = \frac{m}{r/\binom{K}{\eta q} - j} \frac{r/\binom{K}{\eta q} - j}{r/\binom{K}{\eta q}} = \frac{m}{T} \frac{r/\binom{K}{\eta q} - j}{r/\binom{K}{\eta q}}. \tag{11}$$

The number of coded rows per partition that are not yet assigned is given by r/T multiplied by the fraction of partitions removed $\frac{j}{r/\binom{K}{\eta q}}$, i.e.,

$$\frac{1}{T} \frac{rj}{r/\binom{K}{\eta q}} = \frac{1}{T} \frac{m\frac{K}{q}j}{r/\binom{K}{\eta q}}. \tag{12}$$

We assign these rows to batches such that an equal number of coded rows is assigned to each of the K servers, which is always possible due to the limitations imposed by the system model. Any set of q servers will thus store a fraction q/K of these rows. The total number of unique coded rows per partition stored among any set of q servers is then lower bounded by the sum of (12) weighted by q/K and (11), i.e.,

$$\frac{m}{T}\left(\frac{r/\binom{K}{\eta q} - j}{r/\binom{K}{\eta q}} + \frac{\frac{K}{q}j}{r/\binom{K}{\eta q}} \frac{q}{K}\right) = \frac{m}{T},$$

showing that it is possible to decode all partitions using the coded rows stored over any set of q servers.

The communication load is unchanged with respect to the case where the number of partitions is $r/\binom{K}{\eta q}$ if and only if no server receives rows it does not need in the multicast phase. Due to decreasing the number of partitions from $r/\binom{K}{\eta q}$ to $T = r/\binom{K}{\eta q} - j$, we increase the number of coded rows needed to decode each partition by

$$\frac{m}{T} - \frac{m}{r/\binom{K}{\eta q}} = \frac{1}{T} \frac{mj}{r/\binom{K}{\eta q}}. \tag{13}$$

Furthermore, reducing the number of partitions increases the number of coded rows per partition stored among any set of q servers (see (12) and the following text) by

$$\frac{1}{T} \frac{mj}{r/\binom{K}{\eta q}}. \tag{14}$$

Note that the number of additional rows needed to decode each partition (see (13)) is greater than or equal to the number of additional rows stored among the q servers (see (14)). It is thus impossible that too many coded rows are delivered for any partition.

Second, we consider the case $T = \frac{r/\binom{K}{\eta q} - j}{i}$, where j is chosen as for the first case above and where i is a positive integer. Now, we first set all elements of $\boldsymbol{P}$ to i. At this point the number of unique rows of $\boldsymbol{C}$ per partition stored by any set of q servers is given by (11) multiplied by a factor i (since we set each element of $\boldsymbol{P}$ to i instead of one). Furthermore, the number of coded rows per partition that are not yet assigned is given by (12). Therefore, by using the same strategy as for $i = 1$ and assigning the remaining rows to batches such that an equal number of rows is assigned to each of the K servers, we are guaranteed that the communication load and the computational delay are unchanged also in this case. □

5 Assignment Solvers

For $T \leq r/\binom{K}{\eta q}$ partitions, we can choose the assignment matrix $\boldsymbol{P}$ as described in the proof of Theorem 1. For the case where $T > r/\binom{K}{\eta q}$, 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 $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ for all S and $\mathcal{Q} \in \mathbb{Q}^q$. We index each cached $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ by the batches it is computed from. Whenever $U_{\mathcal{Q}}^{(S)}$ drops to 0 due to assigning a row to a batch, we remove the corresponding $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ from the index. We also store a vector of length T with the i-th entry giving the number of vectors $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ that miss intermediate values from the i-th partition. Specifically, the i-th element of this vector is the number of vectors $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ for which the i-th element is less than $\frac{m}{T}$. This allows us to efficiently assess the impact on $L_{\mathbb{Q}}(\boldsymbol{P})$ due to assigning a row to some batch. Since $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ is of length T

Algorithm 1: Heuristic Assignment

```
Input   : P, d, K, T, and ηq
for 0 ≤ a < d(K choose ηq) do
    i ← ⌊a/d⌋ + 1
    j ← (a mod T) + 1
    p_{i,j} ← p_{i,j} + 1
end
return P
```

and because the cardinality of $\mathcal{Q}$ and $\mathbb{Q}^q$ is q and $\binom{K}{q}$, respectively, the memory required to keep this index scales as $\mathcal{O}\left(Tq\binom{K}{q}\right)$ and is thus only an option for small problem instances.

For all solvers, we first label the batches lexiographically and then optimize L_{BDC} in (6). For example, for $\eta q = 2$, we label the first batch by S_1, S_2, the second by S_1, S_3, and so on. The solvers are available under the Apache 2.0 license [25]. We remark that choosing $\boldsymbol{P}$ is similar to the problem of designing the coded matrices stored by each server in [12].

5.1 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

$$Y \triangleq \left\lfloor \frac{r}{\binom{K}{\eta q} \cdot T} \right\rfloor,$$

thus assigning the first $\binom{K}{\eta q}Y$ rows of each partition to batches such that each batch is assigned YT rows. Let $d = r/\binom{K}{\eta q} - YT$ be the number of rows that still need to be assigned to each batch. The $r/T - \binom{K}{\eta q}Y$ rows per partition not assigned yet are assigned in the second step as shown in Algorithm 1.

Interestingly, for $T \leq r/\binom{K}{\eta q}$ the heuristic solver creates an assignment matrix satisfying the requirements outlined in the proof of Theorem 1. In the special case of $T = r/\binom{K}{\eta q}$, the all-ones matrix is produced.

5.2 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 $\binom{K}{\eta q} \times T$ matrix. For each branch, we lower bound 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.

5.2.1 *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 $\boldsymbol{P}$ whose entries do not sum up to the batch size, i.e.,

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

For row i, branch on incrementing the element $p_{i,j}$ by 1 for all columns (with index j) such that their entries do not sum up to the number of coded rows per partition, i.e.,

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

5.2.2 *Bound*

We use a dynamic programming approach to lower bound L_{BDC} for a subtree. Specifically, for each row i and column j of $\boldsymbol{P}$, we store the number of vectors $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ that are indexed by row i and where the j-th element satisfies

$$\left(\frac{m}{T} - \left(\boldsymbol{u}_{\mathcal{Q}}^{(S)}\right)_j\right) > 0.$$

Assigning a coded row to a batch can at most reduce L_{BDC} by $1/\left(mN\left|\mathbb{Q}^q\right|\right)$ for each $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ indexed by that batch. We compute the bound by assuming that no $\boldsymbol{u}_{\mathcal{Q}}^{(S)}$ will be removed from the index for any subsequent assignment.

5.3 Hybrid Solver

The branch-and-bound solver can only be used by itself 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 have entries that sum up 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 entries 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.

6 Luby Transform Codes

In this section, we consider LT codes [16] for use in distributed computing. Specifically, we consider a distributed computing system where $\boldsymbol{\Psi}$ is an LT code encoding matrix, denoted by $\boldsymbol{\Psi}_{\mathsf{LT}}$, of fixed rate $\frac{m}{r}$. As explained in Section 2, we divide the r coded rows of $\boldsymbol{C} = \boldsymbol{\Psi}_{\mathsf{LT}}\boldsymbol{A}$ into $\binom{K}{\eta q}$ disjoint batches, each of which is stored at a unique subset of size ηq of the K servers. For this scheme, due to the random nature of LT codes, we can assign coded rows to batches randomly. The distributed computation is carried out as explained in Section 2.2, i.e., we wait for the fastest $g \geq q$ servers to complete their respective computations in the map phase, perform coded multicasting during the shuffle phase, and carry out the decoding of the N output vectors in the reduce phase.

Let Ω denote the degree distribution and $\Omega(d)$ the probability of degree d. Also, let $\bar{\Omega}$ be the average degree. Then, each row of the encoding matrix $\boldsymbol{\Psi}_{\mathsf{LT}}$ is constructed in the following manner. Uniformly at random select d unique entries of the row, where d is drawn from the distribution Ω. For each of these d entries, assign to it a nonzero element selected uniformly at random from $\mathbb{F}_{2^l}$. Specifically, we consider the case where Ω is the robust Soliton distribution parameterized by M and δ, where M is the location of the spike of the robust component and δ is a parameter for tuning the decoding failure probability for a given M [16].

6.1 Inactivation Decoding

We assume that decoding is performed using inactivation decoding [17]. Inactivation decoding is an efficient maximum likelihood decoding algorithm that combines iterative decoding with optimal decoding in a two-step fashion and is widely used in practice. As suggested in [17], we assume that the optimal decoding phase is performed by Gaussian elimination. In particular, iterative decoding is used until the ripple is empty, i.e., until there are no coded symbols of degree 1, at which point an input symbol is inactivated. The iterative decoder is then restarted to produce a solution in terms of the inactivated symbol. This procedure is repeated until all input symbols are either decoded or inactivated. Note that the value of some input symbols may be expressed in terms of the values of the inactivated symbols at this point. Finally, optimal decoding of the inactivated symbols is performed via Gaussian elimination, and the decoded values are back-substituted into the decoded input symbols that depend on them. The decoding schedule has a large performance impact. Our implementation follows the recommendations in [17]. It is important to tune the parameters M and δ to minimize the number of inactivations.

Due to the nature of LT codes, we need to collect $m(1+\epsilon)$ intermediate values for each vector $\boldsymbol{y}$ before decoding. We refer to ϵ as the overhead. Under inactivation decoding, and for a given overhead ϵ, the probability of decoding failure with an overhead of at most ϵ, denoted by $P_{\mathsf{f}}(\epsilon)$, is lower bounded by [26]

$$P_{\mathsf{f}}(\epsilon) \geq \sum_{i=1}^{m} (-1)^{i+1} \binom{m}{i} \left(\sum_{d=1}^{m} \Omega(d) \frac{\binom{m-i}{d}}{\binom{m}{d}} \right)^{m(1+\epsilon)}. \tag{15}$$

Note that $P_{\mathsf{f}}(\epsilon)$ is the CDF for the random variable "decoding is not possible at a given overhead ϵ." Furthermore, the lower bound (15) well approximates the failure probability for an overhead slightly larger than $\epsilon = 0$. Denote by $F_{\mathsf{DS}}(\epsilon)$ the probability of decoding being possible at an overhead of at most ϵ. It follows that

$$F_{\mathsf{DS}}(\epsilon) = 1 - P_{\mathsf{f}}(\epsilon).$$

We find the decoding success probability density function (PDF) by numerically differentiating $F_{\mathsf{DS}}(\epsilon)$.

6.2 Code Design

We design LT codes for a minimum overhead ϵ_{min}, i.e., we collect at least $m(1+\epsilon_{\mathsf{min}})$ coded symbols from the servers before attempting to decode, and a target failure probability $P_{\mathsf{f,target}} = P_{\mathsf{f}}(\epsilon_{\mathsf{min}})$. We remark that increasing ϵ_{min} and $P_{\mathsf{f,target}}$ leads to a lower average degree $\bar{\Omega}$, and thus to less complex encoding and decoding and subsequently to a lower computational delay for encoding and decoding. The tradeoff is that the communication load increases as more intermediate values need to be transferred over the network on average. Furthermore, increasing ϵ_{min} and $P_{\mathsf{f,target}}$ may increase the average number of servers g required to decode. We thus need to balance the computational delay of the encoding and reduce phases against that

of the map phase to achieve a low overall computational delay. Furthermore, waiting for more than $g = q$ servers typically increases the overall computational delay by more than what is saved by the less complex encoding and decoding given by the larger ϵ_{min} and $P_{\mathsf{f,target}}$. We thus choose ϵ_{min} and $P_{\mathsf{f,target}}$ such that decoding is possible with high probability using the number of coded rows stored at any set of q servers. Note that the overhead ϵ required for decoding may be larger than ϵ_{min}. We take this into account by numerically integrating the decoding success PDF multiplied by the performance of the scheme as a function of the overhead ϵ.

For a given ϵ_{min} and $P_{\mathsf{f,target}}$, we find a pair (M, δ) that minimizes the decoding complexity (see Section 6.3) under the constraint that $P_{\mathsf{f}}(\epsilon_{\mathsf{min}}) \approx P_{\mathsf{f,target}}$. Essentially, we minimize the computational delay of the reduce phase for a fixed delay of the map phase. We remark that LT codes with low decoding complexity have a low average degree $\bar{\Omega}$, and thus also low encoding complexity. Note that for a given M, decreasing δ lowers the failure probability, but also increases the decoding complexity. We find good pairs (M, δ) by selecting through binary search the largest M such that there exists a δ for which the lower bound on $P_{\mathsf{f}}(\epsilon_{\mathsf{min}})$ in (15) is approximately equal to $P_{\mathsf{f,target}}$. This heuristic produces codes with complexity very close to those found using basin-hopping [27] combined with the Powell optimization method [28].

6.3 Computational Delay

There are on average $\bar{\Omega}$ nonzero entries in each row of the LT code encoding matrix. The LT code encoding complexity is thus given by

$$\sigma_{\mathsf{encode,LT}} = \bar{\Omega} rn\sigma_{\mathsf{M}} + (\bar{\Omega} - 1)rn\sigma_{\mathsf{A}}.$$

We simulate the complexity of the decoding $\sigma_{\mathsf{reduce,LT}}$. Furthermore, we assume that the decoding complexity depends only on ϵ_{min}, i.e., we evaluate the decoding complexity only at $\epsilon = \epsilon_{\mathsf{min}}$, and simulate the number of servers g required for a given overhead ϵ.

6.4 Communication Load

The coded multicasting scheme (see Section 2.2.3) is designed for the case where we need m intermediate values per vector $\boldsymbol{y}$. Here, we tune it for the case where we instead need at least $m(1 + \epsilon_{\mathsf{min}})$ intermediate values by increasing the number of coded multicast messages sent. Note that the coded multicasting scheme is greedy in the sense that it starts by multicasting coded messages to the largest possible number of recipients and then gradually lowers the number of recipients. Specifically, we perform the shuffle phase with (see (1))

$$s_{q,\mathsf{LT}} \triangleq \inf\left(s : \sum_{j=s}^{\eta q} \alpha_j \leq (1 + \epsilon_{\mathsf{min}}) - \eta\right).$$

The communication load of the LT code-based scheme for a given $\epsilon \geq \epsilon_{\mathsf{min}}$ is then given by

$$L_{\mathsf{LT}} = \min\left(\sum_{j=s_{q,\mathsf{LT}}}^{\eta q} \frac{\alpha_j}{\phi(j)} + (1+\epsilon) - \eta - \sum_{j=s_{q,\mathsf{LT}}}^{\eta q} \alpha_j \,, \right. \\ \left. \sum_{j=s_{q,\mathsf{LT}}-1}^{\eta q} \frac{\alpha_j}{\phi(j)} + \max\left((1+\epsilon) - \eta - \sum_{j=s_{q,\mathsf{LT}}-1}^{\eta q} \alpha_j, 0\right)\right).$$

6.5 Partitioning of the LT Code-Based Scheme

We can apply partitioning to the LT code-based scheme in the same manner as for the BDC scheme. Specifically, we consider a block-diagonal encoding matrix $\boldsymbol{\Psi}_{\mathsf{BDC-LT}}$, where the blocks $\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_T$ are LT code encoding matrices. In particular, we consider the case where the number of partitions T is equal to the partitioning limit of Theorem 1, i.e., $T = r/\binom{K}{\eta q}$. In this case the all-ones assignment matrix $\boldsymbol{P}$ introduced in the proof of Theorem 1 is a valid matrix. By using this assignment matrix and identical encoding matrices for each of the partitions, i.e., $\boldsymbol{\psi}_i = \boldsymbol{\psi}$, $i = 1, \ldots, T$, the encoding and decoding complexity of each partition is identical regardless of which set of servers $\mathcal{G}$ first completes the map phase. Furthermore, by the same argument as in the proof of Theorem 1, we are guaranteed that if any partition can be decoded using the coded rows stored at the set of servers $\mathcal{G}$, all other partitions can also be decoded.

7 Numerical Results

We present numerical results for the proposed BDC and LT code-based schemes and compare them with the schemes in [7], [8], [9]. Furthermore, we compare the performance of the BDC scheme with assignment $\boldsymbol{P}$ produced by the heuristic and hybrid solvers. We also evaluate the performance of the LT code-based scheme for different $P_{\mathsf{f,target}}$ and ϵ_{min}. For each plot, the field size is equal to one more than the largest number of coded rows considered for that plot, $r + 1$, rounded up to the closest power of 2. The results, except those in Fig. 8 (right), are normalized by the performance of the uncoded scheme. Unless stated otherwise, the assignment $\boldsymbol{P}$ is given by the heuristic solver. As in [9], we assume that $\phi(j) = j$.

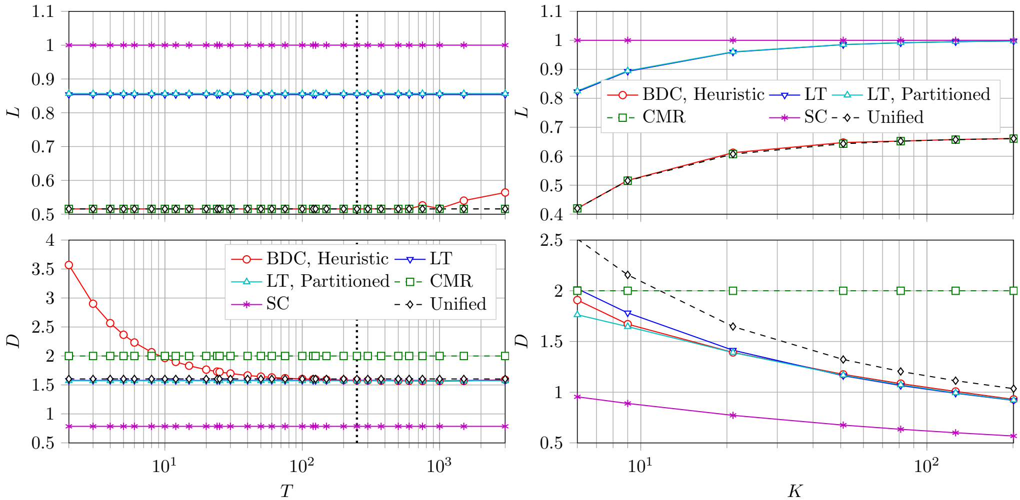


Fig. 5: Left: The tradeoff between partitioning and performance for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6000$, and $\eta = 1/3$. Right: Performance dependence on system size for $\eta q = 2$, $n = m/100$, $\eta m = 2000$, code rate $m/r = 2/3$, and $N = 500q$ vectors.

7.1 Coded Computing Comparison

In Fig. 5 (left), we depict the communication load L (see Definition 1) and the computational delay D (see Definition 2) as a function of the number of partitions, T. The system parameters are $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6000$, and $\eta = 1/3$. The parameters of the CMR and SC schemes are $q_{\mathsf{CMR}} = 9$, $\eta_{\mathsf{CMR}} = \frac{2}{9}$, and $\eta_{\mathsf{SC}} = \frac{1}{6}$. The minimum overhead for the LT code-based scheme is $\epsilon_{\mathsf{min}} = 0.3$ and its target failure probability is $P_{\mathsf{f,target}} = 0.1$. For up to $r/\binom{K}{\eta q} = 250$ partitions (marked by the vertical dotted line), the BDC scheme does not incur any loss in D_{map} and communication load with respect to the unified scheme (see Theorem 1). Furthermore, the BDC scheme yields about a 2% lower delay compared to the unified scheme for $T = 1000$. The delay of the LT code-based scheme is slightly worse than that of the BDC scheme, and the load is about 65% higher (for $T = 250$). Partitioning the LT code-based scheme increases the communication load and reduces the computational delay by about 0.5%. We remark that the number of partitions for the LT code-based scheme is fixed at $r/\binom{K}{\eta q}$. For heavy partitioning of the BDC scheme, a tradeoff between partitioning level, communication load, and map phase delay is observed. For example, with 3000 partitions (the maximum possible), there is about a 10% increase in communication load over the unified scheme. Note that the gain in computational delay saturates, thus there is no reason to partition beyond a given level. The load of the SC scheme is about twice that of our proposed schemes and the delay is about half. Finally, the delay of the BDC and the LT code-based scheme is about 25% lower compared to the CMR scheme for $T > 100$.

In Fig. 5 (right), we plot the performance for a constant $\eta q = 2$, $n = m/100$, $\eta m = 2000$, code rate $m/r = 2/3$, and $N = 500q$ vectors as a function of the number of servers, K. The ratio m/n is motivated by machine learning applications, where the number of rows and columns often represent the number of samples and features, respectively. Note that the number of arithmetic operations performed by each server in the map phase increases with K. We choose the number of partitions T that minimizes the delay under the constraint that the communication load is at most 1% higher compared to the unified scheme. The parameters of the LT code-based scheme are $\epsilon_{\mathsf{min}} = 0.335$ and $P_{\mathsf{f,target}} = 0.1$. The results shown are averages over 1000 randomly generated realizations of $\mathcal{G}$. Our proposed BDC scheme outperforms the unified scheme in terms of computational delay by between about 25% (for $K = 6$) and 10% (for $K = 201$). Furthermore, the delay of both the BDC and LT code-based schemes are about 50% lower than that of the CMR scheme for $K = 201$. For $K = 6$ the computational delay of the unpartitioned and partitioned LT code-based schemes is about 5% higher and 8% lower compared to the BDC scheme, respectively. For $K = 201$ the delay of the LT code-based scheme is about 1% lower than that of the BDC scheme. However, the communication load is about 45% higher. Finally, the communication load of the BDC scheme is between about 42% (for $K = 6$) and 66% (for $K = 201$) of that of the SC scheme.

In Fig. 6 (left), we show the performance for code rate $m/r = 2/3$, $\eta q = 2$, and a fixed workload per server as a function of K. Specifically, we fix the number of additions and multiplications computed by each server in the map phase to 10^8 ($\pm 5\%$ to find valid parameters) and scale m, n, N with K. The number of rows m of $\boldsymbol{A}$ takes values between 12600 and 59800, and we let $n = m/100$ and $N = n$. The number of partitions T is selected in the same way as for Fig. 5 (left). The results shown are averages over 1000

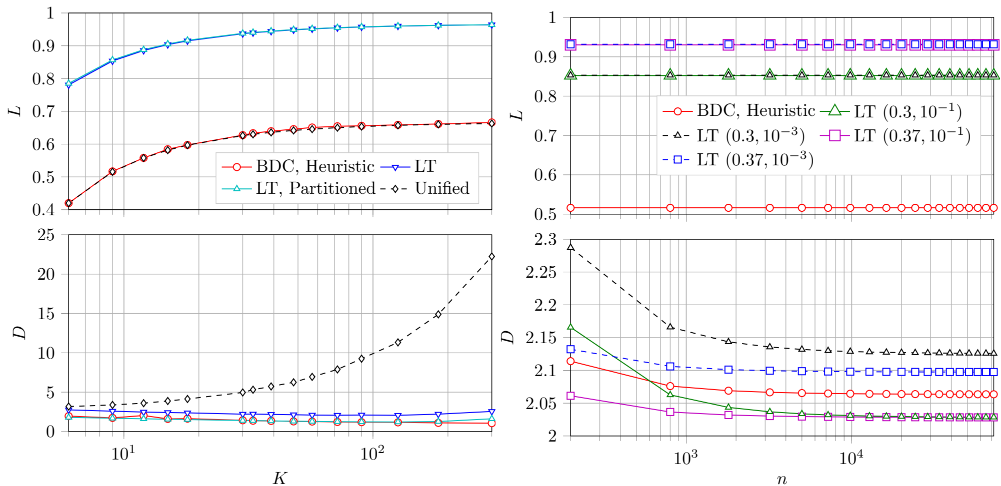


Fig. 6: Left: Performance dependence on system size with constant complexity of the map phase per server, $m/r = 2/3$, $\eta q = 2$, $n = m/100$, and $N = n$. Right: Performance dependence on the number of columns n of $\boldsymbol{A}$ for $m = 2400$, $K = 9$, $q = 6$, $N = 60$, $T = 240$, and $\eta = 1/3$. The parameters of the LT code-based scheme are given in the legend as $(\epsilon_{\mathsf{min}}, P_{\mathsf{f,target}})$.

randomly generated realizations of $\mathcal{G}$. The computational delay of the unified scheme is about a factor 20 higher than that of the BDC scheme for $K = 300$. The computational delay of the partitioned LT code-based scheme is similar to that of the BDC scheme, while the delay of the unpartitioned LT code-based scheme is about 60% higher. Furthermore, the communication load of the LT code-based scheme is about 45% higher compared to those of the unified and BDC schemes.

In Fig. 6 (right), we plot the performance of the BDC and LT code-based schemes as a function of the number of columns n. The system parameters are $m = 2400$, $K = 9$, $q = 6$, $N = 60$, $T = 240$, and $\eta = 1/3$. The communication load of the LT code-based scheme depends primarily on the minimum overhead ϵ_{min} and the computational delay primarily on the target failure probability $P_{\mathsf{f,target}}$. We remark that a higher $P_{\mathsf{f,target}}$ allows for using codes with lower average degree and thus less complex encoding and decoding. For $n = 20000$, the computational delay of the LT code-based scheme with $P_{\mathsf{f,target}} = 0.1$ is about 1.5% lower than that of the BDC scheme. For $P_{\mathsf{f,target}} = 0.001$, the computational delay is about 3% and 1.5% higher than that of the BDC scheme when $\epsilon_{\mathsf{min}} = 0.3$ and $\epsilon_{\mathsf{min}} = 0.37$, respectively. On the other hand, the communication load of the LT code-based scheme with $\epsilon_{\mathsf{min}} = 0.3$ and $\epsilon_{\mathsf{min}} = 0.37$ is about 41% and 44% higher than that of the BDC scheme, respectively.

7.2 Assignment Solver Comparison

In Figs. 7, we plot the performance of the BDC scheme with assignment $\boldsymbol{P}$ given by the heuristic and the hybrid solver. We also give the average performance over 100 random assignments. The vertical dotted line marks the partitioning limit of Theorem 1. The parameters in Fig. 7 are identical to those in Fig. 5.

In Fig. 7 (left), we plot the performance as a function of the number of partitions, T. For T less than about 200, the performance for all solvers is identical. On the other hand, for $T > 200$ both the computational delay and the communication load are reduced with $\boldsymbol{P}$ from the heuristic solver over the random assignments (about 5% for load and 47% 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 Fig. 7 (right), we plot the performance as a function of the number of servers, K. The results shown are averages over 1000 randomly generated realizations of $\mathcal{G}$. For $K = 6$, the communication load of the heuristic solver is about 5% lower than that of the random assignments, but for $K = 201$ the difference is negligible. In terms of computational delay, the heuristic solver outperforms the random assignments by about 18% and 3% for $K = 9$ and $K = 201$, respectively. The hybrid solver is too computationally complex for use with the largest systems considered.

7.3 Tradeoff Between Communication Load and Computational Delay

In Fig. 8 (left), we show the tradeoff between communication load and computational delay. The parameters are $K = 14$, $m = 50000$ ($\pm 3\%$ to find valid parameters), $n = 500$, $N = 840$, and $\eta = 1/2$. Note that the code rate is decreasing toward the bottom of the plot. We select the number of partitions T that minimizes the delay while the load is at most 1% or 10% higher compared to the unified scheme. Allowing a 10% increased load gives up to about 7% lower delay compared to allowing a 1% increase. For the topmost data point

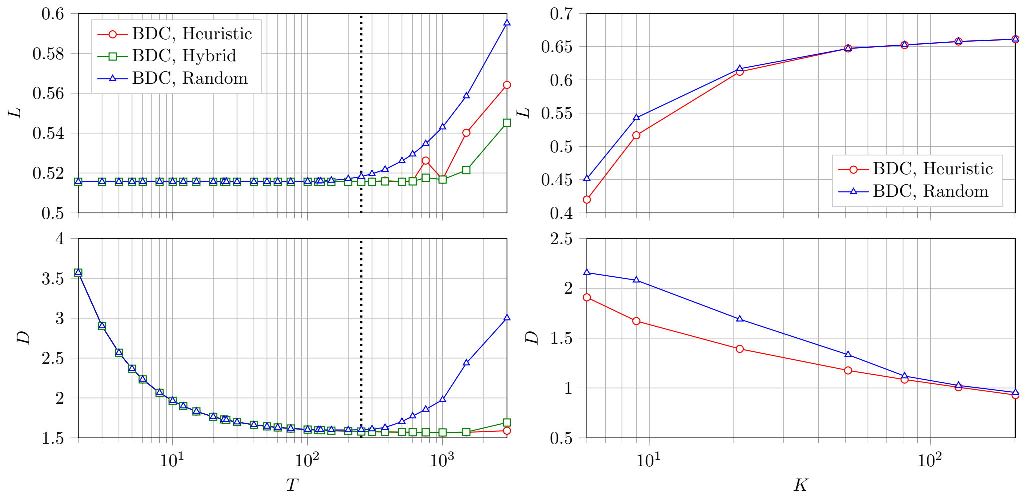


Fig. 7: Left: Solver performance as a function of partitioning for $m = 6000$, $n = 6000$, $K = 9$, $q = 6$, $N = 6000$, and $\eta = 1/3$. Right: Solver performance as a function of system size for $\eta q = 2$, $n = m/100$, $\eta m = 2000$, code rate $m/r = 2/3$, and $N = 500q$ vectors.

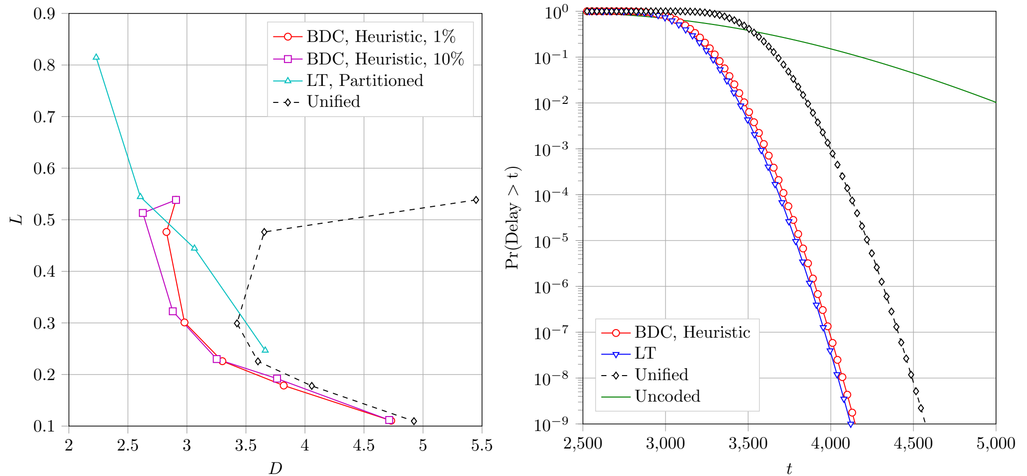


Fig. 8: Left: The tradeoff between communication load and computational delay for $K = 14$, $m = 50000 \pm 3\%$, $n = 500$, $N = 840$, and $\eta = 1/2$. Right: The probability of a computation not finishing before a deadline t for $K = 201$, $q = 134$, $m = 134000$, $n = 1340$, $N = 67000$ vectors, $T = 6700$ partitions, code rate $m/r = 2/3$, $\epsilon_{\text{min}} = 0.335$, and $P_{\text{f,target}} = 10^{-9}$.

of the BDC and unified schemes the encoding complexity dominates, and there is no reason to operate at this point since both the delay and load can be reduced. The parameters of the partitioned LT code-based scheme are $\epsilon_{\text{min}} = 0.3$ and $P_{\text{f,target}} = 10^{-1}$. For the data point with minimum computational delay, the LT code-based scheme yields about 15% lower delay at the expense of about a 30% higher load compared to the BDC scheme. Finally, the computational delay of the BDC scheme is between about 47% and 4% lower compared to the unified scheme for the topmost and bottommost data points, respectively.

7.4 Computational Delay Deadlines

In Fig. 8 (right), we plot the probability of a computation not finishing before a deadline t, i.e., the probability of the computational delay being larger than t. As in [29], we plot the complement of the CDF of the delay in logarithmic scale. On the horizontal axis, we show the deadline t. The system parameters are $K = 201$, $q = 134$, $m = 134000$, $n = 1340$, $N = 67000$ vectors, $T = 6700$ partitions, and code rate $m/r = 2/3$. The parameters for the LT code-based scheme are $\epsilon_{\text{min}} = 0.335$ and $P_{\text{f,target}} = 10^{-9}$. The results

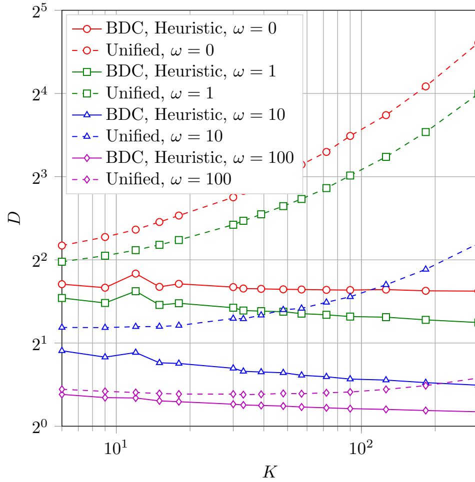


Fig. 9: Computational delay as a function of system size for varying scale of the tail of the runtime distribution. The system parameters and communication load are identical to those in Fig. 6 (left).

are due to simulations. In particular, we simulate the decoding failure probability of LT codes for various t and extrapolate from these points under the assumption that the decoding failure probability is Gamma distributed. The fitted values deviate negligibly from the simulated values.

When the deadline is $t = 3500$, the probability of exceeding the deadline is about 0.4 for the unified and uncoded schemes. For the BDC scheme the probability is only about $7 \cdot 10^{-3}$. The probability is slightly lower for the LT code-based scheme, about $4 \cdot 10^{-3}$. If we instead consider a deadline $t = 4000$, the probability of exceeding the deadline is about 10^{-3} and 0.15 for the unified and uncoded schemes, respectively. For the BDC scheme the probability of exceeding the deadline is about $9 \cdot 10^{-8}$, i.e., 4 orders of magnitude lower compared to the unified scheme. The LT code-based scheme further improves the performance with a probability of exceeding the deadline of about $3 \cdot 10^{-8}$. We remark that for the data point with minimum delay in Fig. 8 (left), the LT code-based scheme has a significant advantage over the BDC scheme in terms of meeting a short deadline.

7.5 Alternative Runtime Distribution

Here, we consider a runtime distribution with CDF

$$F_H(h;\sigma) = \begin{cases} 1 - \mathrm{e}^{-(h-\sigma)/\beta}, & \text{for } h \geq \sigma \\ 0, & \text{otherwise} \end{cases},$$

where σ is the shift and β is a parameter that scales the tail of the distribution, i.e., it differs from the one considered previously by that the scale of the tail may be different from the shift. It is equal to the previously considered distribution if $\beta = \sigma$. This model has been used to model distributed computing in, e.g., [30]. Under this model we assume that the reduce delay of the uncoded scheme follows the distribution above with parameters β and $\sigma_{\mathsf{UC,reduce}} = 0$ since each server has to assemble the final output from the intermediate results regardless coding is used or not. We assume that the encoding delay of the uncoded scheme is zero. Denote by σ_{c} the computational complexity of matrix-vector multiplication for the BDC and unified schemes. We let $\beta = \omega\sigma_{\mathsf{c}}$ for $\omega = 0, 1, 10, 100$. In Fig. 9, we plot the computational delay normalized by that of the uncoded scheme. The system parameters (and thus also the communication load) are identical to those in Fig. 6.

We observe the greatest gain of the BDC scheme over the unified scheme for small ω since the benefits of straggler coding are small compared to the added delay due to encoding and decoding, which is significant for the unified scheme. For larger ω the benefits of straggler coding are larger while the delay due to encoding and decoding remains constant. Hence, the performance of both schemes converge. However, even for $\omega = 100$ the delay of the unified scheme is about 33% higher than that of the BDC scheme for the largest system considered ($K = 300$). We remark that for the example considered in [30] the parameters $\beta = \sigma = 1$, i.e., $\omega = 1$, are used.

8 Conclusion

We introduced two coding schemes for distributed matrix multiplication. One is based on partitioning the matrix into submatrices and encoding each submatrix separately using MDS codes. The other is based on LT

codes. Compared to the earlier scheme in [9] and to the CMR scheme in [7], both proposed schemes yield a significantly lower overall computational delay. For instance, for a matrix of size 59800×598, the BDC scheme reduces the computational delay by about a factor 20 over the scheme in [9] with about a 1% increase in communication load. The LT code-based scheme may reduce the computational delay further at the expense of a higher communication load. For example, for a matrix with about 50000 rows, the computational delay of the LT code-based scheme is about 15% lower than that of the BDC scheme with a communication load that is about 30% higher. Finally, we have shown that the proposed coding schemes significantly increase the probability of a computation finishing within a deadline. The LT code-based scheme may be the best choice in situations where high reliability is needed due to its ability to decrease the computational delay at the expense of the communication load.

ACKNOWLEDGMENT

The authors would like to thank Dr. Francisco Lázaro and Dr. Gianluigi Liva for fruitful discussions and insightful comments on LT codes.

REFERENCES

[1] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
[2] C. L. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, vol. 275, pp. 314–347, Aug. 2014.
[3] L. A. Barroso, "Warehouse-scale computing: The machinery that runs the cloud," in *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2010 Symposium*. Washington, DC: The National Academies Press, 2011, pp. 15–19.
[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Conf. Symp. Operating Systems Design & Implementation*, San Francisco, CA, Dec. 2004, p. 10.
[5] 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, Nov. 2016.
[6] R. Ranjan, "Streaming big data processing in datacenter clouds," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 78–83, May 2014.
[7] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," in *Proc. Allerton Conf. Commun., Control, and Computing*, Monticello, IL, Sep./Oct. 2015, pp. 964–971.
[8] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, Mar. 2018.
[9] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *Proc. Work. Network Coding and Appl.*, Washington, DC, Dec. 2016.
[10] H. Ishii and R. Tempo, "The PageRank problem, multiagent consensus, and web aggregation: A systems and control viewpoint," *IEEE Control Systems Mag.*, vol. 34, no. 3, pp. 34–53, Jun. 2014.
[11] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory*, Aachen, Germany, Jun. 2017, pp. 2418–2422.
[12] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," in *Proc. Advances Neural Inf. Processing Systems*, Long Beach, CA, Dec. 2017, pp. 4403–4413.
[13] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing large linear transforms distributedly using coded short dot products," in *Proc. Advances Neural Inf. Processing Systems*, Barcelona, Spain, Dec. 2016, pp. 2100–2108.
[14] A. Reisizadeh, S. Prakash, R. Pedarsani, and S. Avestimehr, "Coded computation over heterogeneous clusters," in *Proc. IEEE Int. Symp. Inf. Theory*, Aachen, Germany, Jun. 2017, pp. 2408–2412.
[15] A. Severinson, A. Graell i Amat, and E. Rosnes, "Block-diagonal coding for distributed computing with straggling servers," in *Proc. IEEE Inf. Theory Work.*, Kaohsiung, Taiwan, Nov. 2017, pp. 464–468.
[16] M. Luby, "LT codes," in *Proc. IEEE Symp. Foundations Computer Science*, Vancouver, BC, Canada, Nov. 2002, pp. 271–280.
[17] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ Forward Error Correction Scheme for Object Delivery," Internet Requests for Comments, RFC Editor, RFC 6330, Aug. 2011.
[18] J. Edmonds and M. Luby, "Erasure codes with a hierarchical bundle structure," *IEEE Trans. Inf. Theory*, 2017, to appear.
[19] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. European Conf. Computer Systems*, Bordeaux, France, Apr. 2015.
[20] G. Liang and U. C. Kozat, "TOFEC: Achieving optimal throughput-delay trade-off of cloud storage using erasure codes," in *Proc. IEEE Conf. Computer Commun.*, Toronto, ON, Canada, Apr./May 2014, pp. 826–834.
[21] 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, 2008.
[22] C. Walck, "Hand-book on statistical distributions for experimentalists," Particle Physics Group, University of Stockholm, Sweden, Tech. Rep. SUF-PFY/96-01, Sep. 2007. [Online]. Available: http://staff.fysik.su.se/~walck/suf9601.pdf
[23] S.-J. Lin, T. Y. Al-Naffouri, Y. S. Han, and W.-H. Chung, "Novel polynomial basis with fast Fourier transform and its application to Reed-Solomon erasure codes," *IEEE Trans. Inf. Theory*, vol. 62, no. 11, pp. 6284–6299, Nov. 2016.
[24] G. Garrammone, "On decoding complexity of Reed-Solomon codes on the packet erasure channel," *IEEE Commun. Lett.*, vol. 17, no. 4, pp. 773–776, Apr. 2013.
[25] A. Severinson, "Coded Computing Tools," Aug. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1400313
[26] B. Schotsch, G. Garrammone, and P. Vary, "Analysis of LT codes over finite fields under optimal erasure decoding," *IEEE Commun. Lett.*, vol. 17, no. 9, pp. 1826–1829, Sep. 2013.
[27] D. J. Wales and J. P. K. Doye, "Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms," *J. Phys. Chem. A*, vol. 101, no. 28, pp. 5111–5116, Jul. 1997.
[28] M. J. D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *The Computer Journal*, vol. 7, no. 2, pp. 155–162, Jan. 1964.
[29] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," in *Proc. IEEE Int. Symp. Inf. Theory*, Aachen, Germany, Jun. 2017, pp. 2403–2407.
[30] D. Wang, G. Joshi, and G. Wornell, "Using straggler replication to reduce latency in large-scale parallel computing," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 3, pp. 7–11, Dec. 2015.

Paper II

A droplet approach based on raptor codes for distributed computing with straggling servers

Albin Severinson, Alexandre Graell i Amat, Eirik Rosnes, Francisco Lázaro, and Gianluigi Liva

In Proc. IEEE International Symposium on Turbo Codes & Iterative Information Processing.

The layout has been revised.

A Droplet Approach Based on Raptor Codes for Distributed Computing With Straggling Servers

Albin Severinson*†, Alexandre Graell i Amat‡, Eirik Rosnes*,
Francisco Lázaro§, and Gianluigi Liva§

*Simula UiB, Bergen, Norway
†Department of Informatics, University of Bergen, Bergen, Norway
‡Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden
§Institute of Communications and Navigation of DLR (German Aerospace Center), Munich, Germany

Abstract

We propose a coded distributed computing scheme based on Raptor codes to address the straggler problem. In particular, we consider a scheme where each server computes intermediate values, referred to as droplets, that are either stored locally or sent over the network. Once enough droplets are collected, the computation can be completed. Compared to previous schemes in the literature, our proposed scheme achieves lower computational delay when the decoding time is taken into account.

1 Introduction

Modern computing systems often consist of several thousands of servers working in a highly coordinated manner [1]. These systems, referred to as warehouse-scale computers (WSCs) [2], differ from traditional datacenters in that servers rarely have fixed roles. Instead, a cluster manager dynamically assigns storage and computing tasks to servers [1]. This approach offers a high level of flexibility but also poses significant challenges. For example, so-called *straggling servers*, i.e., servers that experience transient delays, are a major issue in WSCs and may significantly slow down the overall computation [3].

Recently, an approach based on maximum distance separable (MDS) codes was proposed to alleviate the straggler problem for linear computations (e.g., multiplying a matrix with a vector) [4], [5]. In particular, redundancy is added to the computation in such a way that straggling servers can be treated as erasures when decoding the final output. Any partially computed results by the straggling servers are discarded. In [4], a single master node is responsible for decoding the final output. A more general framework was proposed in [5], where the work of decoding is distributed over the servers. Somewhat surprisingly, most previous works neglect the decoding complexity of the underlying code, which may have a significant impact on the overall computational delay [6], [7]. For the matrix multiplication problem, a coded scheme consisting of partitioning the source matrix and encoding each partition separately using shorter MDS codes was proposed in [6], [7] and shown to significantly reduce the overall computational delay compared to using a single MDS code when the decoding complexity is taken into account. Furthermore, it was shown in [7] that Luby Transform (LT) codes [8] may reduce the delay further in some cases.

Using LT codes for distributed computing has also been studied in [9], [10], where, assuming that a single master node is responsible for decoding the output, it was shown that these codes may bring some advantages. In [9], the problem of multiplying a matrix by a vector in an internet-of-things setting was considered. Specifically, a scheme based on LT codes where a device may dynamically assign computing tasks to its neighboring devices was proposed. It was shown that this scheme achieves low delay and high resource utilization even when the available computing resources vary over time. The scheme proposed in [10] extends the scheme in [4] by introducing LT codes and utilizing partial computations. The authors give bounds on the overall delay in this setting.

In this paper, we propose a coded computing scheme based on Raptor codes [11] for the problem of multiplying a matrix by a set of vectors. In particular, we consider standardized Raptor10 (R10) codes [12] as the underlying code. Similar to [10], the proposed scheme exploits partial computations, i.e., servers compute intermediate values, referred here to as droplets, that are either stored locally or transferred over the network. The computation can be completed once enough droplets have been collected. Unlike in [4], [5], [9], [10], we take the decoding time into account since it may contribute significantly to the overall computational delay [7]. Furthermore, the work of decoding the output is distributed over the servers in a similar fashion to

the scheme in [5]. We show that this significantly reduces the overall computational delay compared to the scheme in [10] when the number of servers is large, and also outperforms other schemes in the literature. Interestingly, the proposed scheme based on R10 codes achieves an overall computational delay close to that of a scheme using an *ideal* rateless code with zero overhead and incurring no decoding delay. Furthermore, we provide an analytical approximation of the expected overall computational delay of the proposed scheme when the droplets are computed in an optimal order. We then give a heuristic for choosing the order in which each server computes values and show numerically that it achieves almost identical performance to optimal ordering. We also present an optimization problem for finding the optimal number of servers over which the decoding of the final output should be distributed.

2 SYSTEM MODEL AND PRELIMINARIES

We consider the distributed matrix multiplication problem. Specifically, given an $m \times n$ matrix $\boldsymbol{A} \in \mathbb{F}_{2^u}^{m \times n}$ and N vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{F}_{2^u}^{n}$, where $\mathbb{F}_{2^u}$ is an extension field of characteristic 2, 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, $S_1, \ldots, S_K$. More precisely, $\boldsymbol{A}$ is split into m/l disjoint submatrices, each consisting of l rows. The submatrices are then encoded using an $(r/l, m/l)$ linear code, resulting in r/l encoded submatrices, denoted by $\boldsymbol{C}_1, \ldots, \boldsymbol{C}_{r/l}$. We refer to l as the *droplet size*. Each of the r/l coded submatrices is stored at exactly one server such that each server stores ηm coded matrix rows, for some $\frac{1}{K} \leq \eta \leq 1$. Note that, overall, the K servers store a total of $r = \eta m K$ coded rows. We assume that η is selected such that ηm is a multiple of l. Finally, we denote by $\mathcal{C}_k$ the set of indices of the submatrices stored by server S_k.

2.1 Probabilistic Runtime Model

We assume that each server $S_1, \ldots, S_K$ becomes available and starts working on its assigned tasks after a random amount of time, which is captured by the random variables $H_1, \ldots, H_K$, respectively. We assume that $H_1, \ldots, H_K$ are independent and identically distributed (i.i.d) random variables with exponential probability density function

$$f_H(h) = \begin{cases} \frac{1}{\beta} \mathrm{e}^{-\frac{h}{\beta}} & h \geq 0 \\ 0 & h < 0 \end{cases},$$

where β is used to scale the tail of the distribution. The tail accounts for transient disturbances that are at the root of the straggler problem. We refer to β as the straggling parameter. As in [10], we assume that once a server becomes available it carries out each of its assigned tasks in a deterministic amount of time, denoted by σ. Let σ_{A} and σ_{M} be the time required to compute one addition and one multiplication, respectively, over $\mathbb{F}_{2^u}$. The parameter σ is then given by $\sigma = n_{\mathsf{A}}\sigma_{\mathsf{A}} + n_{\mathsf{M}}\sigma_{\mathsf{M}}$, where n_{A} and n_{M} are the required number of additions and multiplications, respectively, to complete each task. As in [12], we assume that σ_{A} is $\mathcal{O}(\frac{u}{64})$ and σ_{M} is $\mathcal{O}(u \log_2 u)$. Furthermore, we assume that the hidden coefficients are comparable and will thus not consider them.

We denote by $H_{(i)}$, $i = 1, \ldots, K$, the i-th order statistic, i.e., the i-th smallest variable of $H_1, \ldots, H_K$. $H_{(i)}$ is a gamma-distributed random variable with cumulative probability distribution function

$$F_{H_{(i)}}(h_{(i)}) \triangleq \Pr(H_{(i)} \leq h_{(i)}) = \begin{cases} \frac{\gamma(b, a h_{(i)})}{\Gamma(b)} & h_{(i)} \geq 0 \\ 0 & h_{(i)} < 0 \end{cases},$$

where Γ denotes the gamma function and γ the lower incomplete gamma function. The inverse scale factor a and shape parameter b of the gamma distribution are computed from its mean and variance as in [7]. The expectation of $H_{(i)}$, i.e., the expected delay until a total of i servers become available, is [13]

$$\mu(K, i) \triangleq \mathbb{E}\left[H_{(i)}\right] = \sum_{j=K-i+1}^{K} \frac{\beta}{j}.$$

Finally, we denote by h_i and $h_{(i)}$ the realizations of H_i and $H_{(i)}$, $i = 1, \ldots, K$, respectively.

2.2 Distributed Computing Model

We consider the coded computing framework introduced in [5], which extends the MapReduce framework [3]. The overall computation proceeds in two phases, the *map-shuffle* phase and the *reduce* phase, which are augmented to make use of the coded scheme proposed in [10] 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.

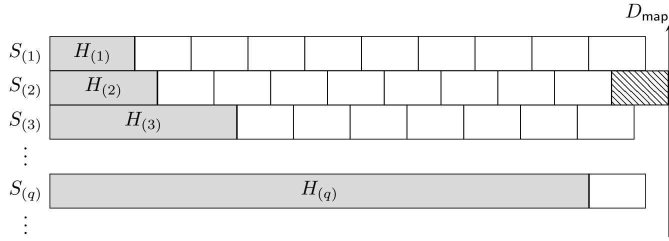


Fig. 1: Map-shuffle phase computation. Each server $S_{(k)}$, $k = 1, \dots, K$, computes droplets, illustrated by white squares, after an initial time $H_{(k)}$. The map-shuffle phase ends once enough droplets are collected and server $S_{(q)}$ has become available. It incurs a delay D_{map}. We depict the final droplet that is computed with a hash pattern.

2.2.1 *Map-Shuffle Phase*

The servers compute coded intermediate values (droplets) which are later used to obtain the vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$. Each droplet is the product between a submatrix stored by the server and an input vector $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$. The responsibility for decoding each of the vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ is assigned to one of the K servers. The computed droplets are then transferred over the network to the server responsible for decoding the corresponding output vector. We assume that the channel is error-free and that all transfers are unicast. The map-shuffle phase ends when all output vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ can be decoded with high probability (see Section 3.2). At this point the computation enters the reduce phase. We denote the delay of the map-shuffle phase by D_{map} and its expectation by $\bar{D}_{\mathsf{map}}$.

2.2.2 *Reduce Phase*

The vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ are computed from the intermediate values. More specifically, each server uses the droplets computed locally or received over the network to decode the output vectors it has been assigned. Denote by σ_{reduce} the time required for one server to decode one output vector. The computational delay of the reduce phase, denoted by $\mathsf{D}_{\mathsf{reduce}}$, is deterministic and is given by $\mathsf{D}_{\mathsf{reduce}} = \frac{N}{q}\sigma_{\mathsf{reduce}}$, where q denotes the number of servers used in the reduce phase.

Definition 1. *The overall computational delay, D, is the sum of the map-shuffle and reduce phase delays, i.e.,*

$$D = D_{\mathsf{map}} + \mathsf{D}_{\mathsf{reduce}} \quad \text{and} \quad \bar{D} \triangleq \mathbb{E}[D] = \bar{D}_{\mathsf{map}} + \mathsf{D}_{\mathsf{reduce}}.$$

2.3 Raptor Codes

Raptor codes [11] are built from the serial concatenation of an outer linear block code with an inner LT code. Raptor codes not only outperform LT codes in terms of probability of decoding failure but also exhibit a lower encoding and decoding complexity. Here we consider R10 codes, which are binary codes whose outer code is obtained as the serial concatenation of a low-density parity-check code with a high-density parity-check (HDPC) code [14]. R10 codes are tailored to an efficient maximum likelihood decoding algorithm known as *inactivation decoding* [11]. In particular, we consider R10 codes in their nonsystematic form.

3 Proposed Coded Computing Scheme

In this section, we introduce the proposed coded computing scheme. The main idea is that each server computes multiple intermediate values. More specifically, each server S_k, $k = 1, \dots, K$, computes droplets $\boldsymbol{z}_j^{(i)} = \boldsymbol{C}_i \boldsymbol{x}_j$ by multiplying the coded submatrices $\boldsymbol{C}_i$, $i \in \mathcal{C}_k$, it stores locally with the N input vectors $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$. The indices $i \in \mathcal{C}_k$ and $j \in \{1, \dots, N\}$ should be carefully chosen to minimize the computational delay. We consider this in Section 3.1. The time required for a server to compute a droplet, denoted by σ_{d}, is $\sigma_{\mathsf{d}} = l\left((n-1)\sigma_{\mathsf{A}} + n\sigma_{\mathsf{M}}\right)$ since it requires computing l inner products, each requiring $n-1$ additions and n multiplications.

Denote by $S_{(1)}$ the first server to become available, and similarly denote by $S_{(k)}$, $k = 1, \dots, K$, the k-th server to become available. We assume that server $S_{(k)}$ computes droplets at a constant rate after a delay $H_{(k)}$. For example, server $S_{(k)}$ computes p droplets after a total delay of $H_{(k)} + p\sigma_{\mathsf{d}}$. This process is depicted in Fig. 1. We evenly and randomly split the indices of the N output vectors $\boldsymbol{y}_1, \dots, \boldsymbol{y}_N$ into $q \leq K$ disjoint sets $\mathcal{W}_1, \dots, \mathcal{W}_q$. Each of the q fastest servers $S_{(k)}$, $k = 1, \dots, q$, is responsible for decoding the N/q output vectors with indices in $\mathcal{W}_k$. Furthermore, we denote by $\tilde{\mathcal{W}}_k$ the set containing the indices of the vectors that server $S_{(k)}$ is not yet able to decode due to an insufficient number of droplets. At the start of the map-shuffle phase, $\tilde{\mathcal{W}}_k = \mathcal{W}_k$. The map-shuffle phase ends when servers $S_{(1)}, \dots, S_{(q)}$ have collected enough droplets to decode the output vectors they are responsible for, i.e., when $\left|\tilde{\mathcal{W}}_k\right| = 0$, $k = 1, \dots, q$. At this point servers $S_{(1)}, \dots, S_{(q)}$ simultaneously enter the reduce phase. The remaining $K - q$ servers are unused for the rest of the computation. A strategy for choosing q to minimize the expected computational delay, $\bar{D}$, is discussed in Section 4.1.

3.1 Droplet Order

For each droplet $\boldsymbol{z}_j^{(i)}$ computed by server S_k in the map-shuffle phase, the server has to choose the indices $i \in \mathcal{C}_k$ and $j \in \{1, \dots, N\}$ the droplet is computed from. Furthermore, the choice of i and j may have a large impact on the computational delay. In particular, if j is chosen such that it is not needed to decode any of the output vectors, i.e., j is not in any of the sets $\tilde{\mathcal{W}}_k$, $k = 1, \dots, q$, the resulting droplet is effectively wasted. Hence, i and j should be carefully chosen. We consider two scenarios. In the first scenario, i and j are chosen optimally, i.e., all servers have perfect knowledge of $\tilde{\mathcal{W}}_1, \dots, \tilde{\mathcal{W}}_q$. This gives a lower bound on the achievable computational delay. In a second, more practical scenario, i and j are chosen in a round-robin fashion. Specifically, for each server S_k we generate a number j from $\{1, \dots, N\}$ uniformly at random. Next, for each droplet $\boldsymbol{z}_j^{(i)}$ computed by server S_k we let $j = j + 1 \bmod N$. We remark that the optimal order requires each server to have global knowledge of all previously computed droplets over all servers, whereas the round-robin strategy only requires each server to have knowledge of the droplets it has computed locally. In Section 5, we show numerically that the round-robin strategy achieves almost identical performance to the optimal strategy, the latter being infeasible in practice. In both cases we assume that the same pair of indices i, j is never chosen twice. Since each submatrix $\boldsymbol{C}_i$ is stored at exactly one server, this does not require any additional synchronization between servers. A server that has exhausted all possible combinations of i and j halts and performs no further computations in the map-shuffle phase.

3.2 Code Design

The decoding complexity and failure probability of Raptor codes depend on the number of droplets available to the decoder, $\frac{m}{l}(1+\epsilon)$, for some $\epsilon \geq 0$. We refer to ϵ as the overhead. Furthermore, we denote by $P_{\mathsf{f}}(\epsilon)$ the decoding failure probability when the overhead is ϵ. In general, increasing ϵ reduces the probability of decoding failure $P_{\mathsf{f}}(\epsilon)$ and the decoding complexity, leading to a lower decoding time σ_{reduce}. For example, the decoding failure probability for R10 codes roughly halves with every additional droplet available when the number of source symbols is close to 1000 [11]. However, a larger overhead ϵ also increases the computational delay due to computing the required droplets in the map-shuffle phase. We thus need to balance the computational delay of the reduce phase against that of the map-shuffle phase to achieve a low overall computational delay.

We denote by ϵ_{min} the minimum overhead before decoding is attempted. R10 codes are fully specified, hence the only free parameter is ϵ_{min}. In [11], it is observed that the decoding complexity of Raptor codes drops sharply when the number of droplets available to the decoder is increased to be slightly larger than the number of HDPC symbols. Hence, we choose the minimum overhead ϵ_{min} such that the number of droplets available to the decoder is close to the number of source droplets m/l plus twice the number of HDPC symbols. For comparison purposes, in Section 5 we also consider LT codes with a robust Soliton distribution [8], whose parameters are optimized as described in [7]. In particular, we choose a minimum overhead ϵ_{min} and a target failure probability $P_{\mathsf{f,target}}$ and optimize the parameters of the distribution to minimize the decoding complexity under the constraint $P_{\mathsf{f,target}} \approx P_{\mathsf{f}}(\epsilon_{\mathsf{min}})$. Note that the overhead ϵ required for decoding may be larger than ϵ_{min}. We take this into account by simulating the overhead needed given that decoding failed at an overhead of ϵ_{min}.

4 Computational Delay Analysis

In this section, we analyze the computational delay of the proposed coded computing scheme and provide an approximation of $\bar{D}_{\mathsf{map}}$. Let V_p be the random variable associated with the time until p droplets are computed over K servers, where we assume that p is chosen such that decoding succeeds with high probability, and $\bar{V}_p$ its expectation, $\bar{V}_p \triangleq \mathbb{E}[V_p]$. Then, $D_{\mathsf{map}} = \max(V_p, H_{(q)})$. For the analysis, we assume that each server is always able to compute droplets needed by some server until the end of the map-shuffle phase. This assumption is valid if the code rate m/r is low enough. Furthermore, we assume that the droplet order is optimal (see Section 3.1). Finally, we explain how to choose the number of servers q to split the output vectors over to minimize the expected computational delay.

Denote by P_t the number of droplets computed over K servers at time t.

Proposition 1. *The expectation of P_t is*

$$\bar{P}_t \triangleq \mathbb{E}[P_t] = K \int_0^t \left\lfloor \frac{t-h}{\sigma_{\mathsf{d}}} \right\rfloor \frac{1}{\beta} \mathrm{e}^{-\frac{h}{\beta}} \, \mathrm{d}h. \tag{1}$$

Using the fact that $x - 1 \leq \lfloor x \rfloor \leq x$ in (1) and computing the resulting integrals, $\bar{P}_t$ can be lower and upperbounded as

$$K\left(\frac{(\beta+\sigma_{\mathsf{d}})\mathrm{e}^{-\frac{t}{\beta}}}{\sigma_{\mathsf{d}}} + \frac{t}{\sigma_{\mathsf{d}}} - \frac{\beta}{\sigma_{\mathsf{d}}} - 1\right) \leq \bar{P}_t \leq K\left(\frac{\beta \mathrm{e}^{-\frac{t}{\beta}}}{\sigma_{\mathsf{d}}} + \frac{t}{\sigma_{\mathsf{d}}} - \frac{\beta}{\sigma_{\mathsf{d}}}\right). \tag{2}$$

Let $\sigma_{\bar{P}}$ denote the time at which an average number of $\bar{P}$ droplets have been computed over K servers. By inverting the upper and lower bounds on $\bar{P}_t$ in (2), $\sigma_{\bar{P}}$ can be bounded as

$$\sigma_{\bar{P}}^{\mathsf{L}} \triangleq \beta + \frac{\bar{P}\sigma_{\mathsf{d}}}{K} + \beta W_0\left(-\mathrm{e}^{-\frac{\bar{P}\sigma_{\mathsf{d}}}{K\beta}-1}\right) \leq \sigma_{\bar{P}} \leq \beta + \sigma_{\mathsf{d}} + \frac{\bar{P}\sigma_{\mathsf{d}}}{K} + \beta W_0\left(-\frac{\mathrm{e}^{-\frac{K(\beta+\sigma_{\mathsf{d}})+\bar{P}\sigma_{\mathsf{d}}}{K\beta}}(\beta+\sigma_{\mathsf{d}})}{\beta}\right) \triangleq \sigma_{\bar{P}}^{\mathsf{U}},$$

where $W_0(\cdot)$ is the principal branch of the Lambert W function, i.e., $W_0(x)$ is the solution of $x = z\mathrm{e}^z$.

Now, let G_t be the random variable associated with the number of servers that are available at time t. We provide the following heuristic approximation of $\bar{D}_{\mathsf{map}}$,

$$\bar{D}_{\mathsf{map}} \approx \bar{V}_p + \sum_{j=1}^{q-1} \Pr(G_t = j)\mu(K-j, q-j), \tag{3}$$

where the summation accounts for the delay due to waiting for server $S_{(q)}$. We have numerically verified that the approximation holds. Furthermore, we have observed that $\bar{V}_p \approx \sigma_p$ and $\sigma_p \approx \frac{1}{2}\left(\sigma_p^{\mathsf{L}} + \sigma_p^{\mathsf{U}}\right)$. Finally, assuming that decoding is possible with p droplets, the expected overall computational delay is

$$\bar{D} \approx \frac{N}{q}\sigma_{\mathsf{reduce}} + \bar{V}_p + \sum_{j=1}^{q-1} \Pr(G_t = j)\mu(K-j, q-j). \tag{4}$$

4.1 Straggler Mitigation

The map-shuffle phase ends when all output vectors can be decoded and when the servers $S_{(1)}, \ldots, S_{(q)}$ are available, i.e., $D_{\mathsf{map}} = \max(V_p, H_{(q)})$. Since $\Pr(H_{(q)} > V_p)$ is always nonzero, choosing a small q lowers the expected delay of the map phase. On the other hand, choosing a large q reduces the delay of the reduce phase $\mathsf{D}_{\mathsf{reduce}} = \frac{N}{q}\sigma_{\mathsf{reduce}}$, as the decoding is distributed over more servers. Thus, we need to balance the delay of the map-shuffle and reduce phases by choosing q carefully. In particular, we optimize the value of q to minimize the overall computational delay in (4), where we use the approximation $\bar{V}_p \approx \sigma_p \approx \frac{1}{2}\left(\sigma_p^{\mathsf{L}} + \sigma_p^{\mathsf{U}}\right)$. We remark that (4) as a function of q is convex as it is the sum of the approximation of $\bar{D}_{\mathsf{map}}$ in (3) and $\mathsf{D}_{\mathsf{reduce}}$, which are strictly increasing and decreasing, respectively, in q for $\sigma_{\mathsf{reduce}} > 0$. For $\sigma_{\mathsf{reduce}} = 0$, (4) is minimized for $q = 1$.

5 Numerical Results

In Fig. 2 (left), we give the expected computational delay of the proposed scheme, normalized by that of the uncoded scheme, as a function of the system size. In particular, we fix the code rate to $m/r = 1/3$ and the problem size divided by the number of servers to $mnN/K = 10^7$ ($\pm 10\%$ to find valid parameters) and scale the system size with K. Motivated by machine learning applications, where the number of rows and columns often represent the number of samples and features, respectively, we set $m = 1000n$. We also set $N = 10K$. Since R10 codes are optimized for code lengths close to 1024 [14], we choose the droplet size l such that $900 < m/l < 1100$ (the interval is required to find valid parameters). The overhead is 2% and 30% for R10 and LT codes, respectively. Finally, the straggling parameter β is equal to the total time required to compute the multiplications $\boldsymbol{A}\boldsymbol{x}_1, \ldots, \boldsymbol{A}\boldsymbol{x}_N$ divided by the number of servers, i.e., $\beta = \sigma_{\mathsf{K}} = (m(n-1)\sigma_{\mathsf{A}} + mn\sigma_{\mathsf{M}})N/K$.

In the figure, we plot the overall computational delay given by (4) using the approximation $\bar{V}_p \approx \sigma_p \approx \frac{1}{2}\left(\sigma_p^{\mathsf{L}} + \sigma_p^{\mathsf{U}}\right)$ for the proposed scheme with an underlying R10 code (blue line with circle markers) and LT code (magenta line with diamond markers), and for the scheme assuming an ideal rateless code (black solid line). We also show simulated performance for the R10-based scheme with optimal droplet ordering and with a round-robin (rr) ordering. We observe that the round-robin strategy achieves a computational delay within 1% of that of the optimal strategy. Furthermore, (4) accurately predicts the overall computational delay with an error of at most about 1% compared to both the optimal and the round-robin ordering. The proposed scheme with R10 codes achieves a significantly lower delay than the scheme with LT codes. Interestingly, the delay for the scheme based on R10 codes is very close (at most 3.7% higher) to that of an ideal rateless code.

For comparison purposes, we also plot in the figure the delay of the block-diagonal coding (BDC) scheme in [6], [7], the MDS coding scheme proposed in [4] that does not utilize partial computations, and the scheme proposed in [10] (augmented with R10 codes). We refer to it as the centralized R10 (cent. R10) scheme, since a central master node is responsible for decoding all output vectors. For small K, the delay is limited by the time needed to compute droplets. However, for $K \gtrsim 90$ the master node of the centralized scheme can no longer decode the output vectors quickly enough, causing a high overall computational delay. Thus, for $K \gtrsim 90$ the scheme in [10] (now with R10 codes), incurs a delay significantly higher than that of the proposed scheme. The proposed scheme also yields a significantly lower computational delay than that of the scheme in [4]. Finally, the delay of the BDC scheme in [6], [7] is about 10% higher compared to the proposed scheme based on R10 codes.

In Fig. 2 (right), we give the expected computational delay as a function of the straggling parameter β for $K = 625$, $m = 33333$, $n = 33$, $N = 6250$, $l = 32$, and $m/r = 1/3$. Since l is not a divisor of m, $\boldsymbol{A}$ is zero-padded with 11 all-zero rows. The performance of the centralized scheme approaches that of our scheme as β grows since the average rate at which droplets are computed decreases with β. The scheme based on R10 codes operates close to an ideal rateless code for all values of β considered.

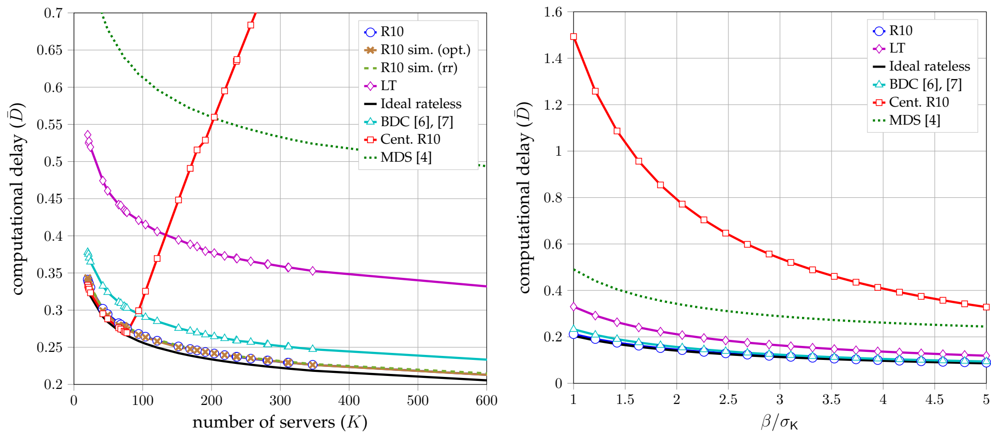


Fig. 2: Left: Performance dependence on system size for $mnN/K \approx 10^7$, $n = m/1000$, $N = 10K$, $m/r = 1/3$, $m/l \approx 1024$, and $\beta = \sigma_K$. Right: Performance dependence on the straggling parameter β for $K = 625$, $m = 33333$, $n = 33$, $N = 6250$, $l = 32$, and $m/r = 1/3$.

6 Conclusion

We introduced a coded computing scheme based on Raptor codes for distributed matrix multiplication where each server computes several intermediate values and where the work of decoding the output is distributed among servers. Compared to previous schemes, the proposed scheme yields significantly lower computational delay when the number of servers is large. For instance, the delay is less than half when the number of servers is 200. Furthermore, the performance of the scheme based on R10 codes is close to that of an ideal rateless code.

References

[1] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. European Conf. Computer Systems*, Bordeaux, France, Apr. 2015.

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

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Conf. Symp. Operating Systems Design & Implementation*, San Francisco, CA, Dec. 2004, p. 10.

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

[5] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *Proc. Work. Network Coding and Appl.*, Washington, DC, Dec. 2016.

[6] A. Severinson, A. Graell i Amat, and E. Rosnes, "Block-diagonal coding for distributed computing with straggling servers," in *Proc. IEEE Inf. Theory Work.*, Kaohsiung, Taiwan, Nov. 2017, pp. 464–468.

[7] ——, "Block-diagonal and LT codes for distributed computing with straggling servers," Dec. 2017. [Online]. Available: https://arxiv.org/abs/1712.08230v2

[8] M. Luby, "LT codes," in *Proc. IEEE Symp. Foundations Computer Science*, Vancouver, BC, Canada, Nov. 2002, pp. 271–280.

[9] Y. Keshtkarjahromi, Y. Xing, and H. Seferoglu, "Dynamic heterogeneity-aware coded cooperative computation at the edge," Jan. 2018. [Online]. Available: https://arxiv.org/abs/1801.04357v2

[10] A. Mallick, M. Chaudhari, and G. Joshi, "Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication," Apr. 2018. [Online]. Available: https://arxiv.org/abs/1804.10331v2

[11] A. Shokrollahi and M. Luby, "Raptor codes," *Foundations and Trends in Commun. and Inf. Theory*, vol. 6, no. 3–4, pp. 213–322, May 2011.

[12] J. Edmonds and M. Luby, "Erasure codes with a hierarchical bundle structure," *IEEE Trans. Inf. Theory*, 2017, to appear.

[13] 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, 2008.

[14] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer, "Raptor Forward Error Correction Scheme for Object Delivery," Internet Requests for Comments, RFC Editor, RFC 5053, Oct. 2007.

Paper III

DSAG: A mixed synchronous-asynchronous iterative method for straggler-resilient learning

Albin Severinson, Eirik Rosnes, Salim El Rouayheb, and Alexandre Graell i Amat

Under review for potential publication in IEEE Transactions on Cloud Computing.

The layout has been revised.

DSAG: A mixed synchronous-asynchronous iterative method for straggler-resilient learning

Albin Severinson*†, Eirik Rosnes*, Salim El Rouayheb§, and Alexandre Graell i Amat‡

*Simula UiB, Bergen, Norway
†Department of Informatics, University of Bergen, Bergen, Norway
§Department of Electrical and Computer Engineering, Rutgers University, Piscataway, New Jersey
‡Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden

Abstract

We consider straggler-resilient learning. In many previous works, e.g., in the coded computing literature, straggling is modeled as random delays that are independent and identically distributed between workers. However, in many practical scenarios, a given worker may straggle over an extended period of time. We propose a latency model that captures this behavior and is substantiated by traces collected on Microsoft Azure, Amazon Web Services (AWS), and a small local cluster. Building on this model, we propose DSAG, a mixed synchronous-asynchronous iterative optimization method, based on the stochastic average gradient (SAG) method, that combines timely and stale results. We also propose a dynamic load-balancing strategy to further reduce the impact of straggling workers. We evaluate DSAG for principal component analysis, cast as a finite-sum optimization problem, of a large genomics dataset, and for logistic regression on a cluster composed of 100 workers on AWS, and find that DSAG is up to about 50% faster than SAG, and more than twice as fast as coded computing methods, for the particular scenario that we consider.

1 Introduction

We are interested in reducing the latency of distributed iterative optimization methods for empirical risk minimization. In particular, we want to reduce the impact of straggling workers, i.e., workers experiencing delays, which can significantly slow down distributed algorithms. The straggler problem is a consequence of the design of modern large-scale compute clusters (sometimes referred to as *warehouse-scale computers*), which are built from a large number of commodity servers connected in a heterogeneous manner, and where many virtual machines may share the same physical host server, to maximize cost-efficiency [1], [2]. Examples include Microsoft Azure, Google Cloud, and Amazon Web Services (AWS).

Straggling is often assumed, e.g., in the coded computing literature [3], [4], [5], [6], to be caused by random delays that are independent and identically distributed (i.i.d.) between workers and iterations. However, from traces collected on Microsoft Azure and AWS, we find that stragglers tend to remain stragglers. As a result, data processed by stragglers may never factor in for stochastic methods that only rely on the results from the fastest subset of workers.

This work consists of three parts. First, we propose a latency model that, unlike previous models, accounts for differences in the mean and variance of the latency between different workers and over time. Further, for the proposed model we show how to efficiently estimate the latency of the w-th fastest worker out of a set of N workers, including for iterative computations, where a worker may remain unavailable over several subsequent iterations.

Second, based on this model, we propose DSAG, an iterative method for finite-sum optimization (machine learning problems are typically cast as finite-sum optimization problems) which adapts the stochastic average gradient (SAG) method [7] to distributed environments. The key idea of DSAG is to wait for the w fastest workers in each iteration—i.e., DSAG is a stochastic method—while simultaneously integrating stale results received from the $N - w$ stragglers as they are received over subsequent iterations. DSAG relies on the *variance reduction* technique of SAG to suppress the potentially high variance caused by this strategy and improve convergence. Finally, we propose a dynamic load-balancing strategy for reducing the variation in latency between workers, that is based on the model proposed in part one.

We validate the proposed model on Azure, AWS, and a small local cluster, and find that the model accurately predicts latency across the three platforms. We evaluate the performance of DSAG by using it for principal component analysis (PCA), cast as an optimization problem, of a large genomics dataset, and for logistic regression. For both PCA and logistic regression, DSAG with load balancing reduces latency

significantly compared to SAG—for a scenario with 100 workers on AWS, DSAG is about 10% faster than SAG for PCA and up to 50% faster for logistic regression. Furthermore, it is more than twice as fast as coded computing methods.

We provide the source code of our implementation and the latency traces we have collected under a permissive license at [8].

Related work

Recently, *coded computing* has been proposed to deal with stragglers [3]. The key idea is to add redundant computations (thus increasing the per-worker computational load), such that the result of the computation can be recovered from a subset of the workers, typically via a decoding operation. Coded computing methods have been proposed for, e.g., matrix-vector multiplication [3], [9], [10], matrix-matrix multiplication [11], [12], [13], [14], [15], [16], [17], [18], polynomial evaluation [19], and gradient computations [4], [20]. For example, the method in [4] increases the computational load per worker by a factor $(N - w) + 1$ compared to gradient descent (GD) to tolerate any $N - w$ stragglers.

Another method to deal with stragglers is stochastic optimization, the simplest form of which is to ignore stragglers for GD. This is a stochastic gradient descent (SGD) method, sometimes referred to as ignoring stragglers SGD. SGD does not converge to the optimum unless the stepsize is reduced as the algorithm progresses. However, a smaller stepsize reduces the rate of convergence, and it is difficult to determine the correct rate at which to reduce the stepsize. Approximate coded computing methods combine ignoring stragglers SGD with redundancy, e.g., [5], [21], [22]. These methods improve the rate of convergence per iteration compared to ignoring stragglers SGD but typically do not converge to the optimum, and typically increase the computational load compared to GD by a factor 2 or 3.

The above methods treat iterations independently, ignoring the correlation between the results computed in subsequent iterations, which is often significant. The coded version of the power method proposed in [6] is an exception in that the previous iterate is used as side information during decoding. The process is related to *sketch-and-project* methods (see, e.g., [23], [24], [25]), i.e., iterative methods to approximate some quantity from low-rank sketches. In particular, the method in [6] can be seen as a special case of the one in [24]. A significant shortcoming of the method in [6] is that it requires a complex decoding process to be performed by the coordinator for each iteration.

The method in [24] is a variance-reduced stochastic method for first-order optimization. For each iteration, these methods use an estimate of the gradient to, e.g., perform a gradient step, i.e., they are stochastic. Variance-reduced methods converge to the optimum despite being stochastic by using information contained in previous iterates and/or gradients to ensure that the variance of this estimate tends to zero as the method progresses. Examples of variance-reduced methods include SAG [7], SAGA [26] (including a peer-to-peer version [27]), SARAH [28], SVRG [29], SEGA [24], and MARINA [30]. These works do not consider the straggler problem.

Exploiting stale gradients in combination with asynchronicity to alleviate the straggler problem has been explored in several previous works, see, e.g., [31], [32], and references therein, in the neighboring area of federated learning, and [13]. These methods are similar to the proposed DSAG, but do not employ variance reduction. For example, a mixed synchronous-asynchronous distributed version of SGD that is similar to ours has been proposed and analyzed in [13]. Like the method we propose, the method in [13] uses asynchronicity to reduce iteration latency. However, unlike our method, the method in [13] gradually increases the level of synchronicity, thus increasing iteration latency, to improve convergence, whereas our method relies on variance reduction. There has also been a significant amount of work on asynchronous optimization for shared-memory systems, e.g., [33], [34], and references therein. However, these works do not consider the straggler problem.

The load-balancing approach we suggest is designed specifically for DSAG, but is inspired by the large number of previous works on the topic; see, e.g., [35], [36], [37], [38], and references therein. These suggest approaches to balance either i) the complexity of the subtasks that make up a particular large computation (e.g., [35], [36]), or ii) incoming requests between instances of a distributed application, such as a web server (e.g., [37], [38]). The approach we suggest, like those of [37], [38], but unlike [35], [36], accounts for latency differences between servers and over time—as is the case in the cloud—but balances the complexity of subtasks, as in [35], [36]. Furthermore, DSAG is designed with load-balancing in mind, and, as a result, unlike the approach of [35], [36], does not require moving data between servers to perform load-balancing.

2 Preliminaries

Denote by $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ a data matrix, where n is the number of samples and d the dimension. Many learning problems (e.g., linear and logistic regression, PCA, matrix factorization, and training neural networks) can be cast as a finite-sum optimization problem of the form

$$\boldsymbol{V}^* = \underset{\boldsymbol{V} \in \mathcal{L}}{\arg\min} \left[F(\boldsymbol{V}, \boldsymbol{X}) \triangleq R(\boldsymbol{V}) + \sum_{i=1}^{n} f_i(\boldsymbol{V}, \boldsymbol{x}_i) \right], \quad (1)$$

where $\mathcal{L}$ is the solution space, f_i is the loss function with respect to the i-th sample (row of $\boldsymbol{X}$), which we denote by $\boldsymbol{x}_i$, and R is a regularizer, which serves to, e.g., bias $\boldsymbol{V}^*$ toward sparse solutions. For the remainder of this paper, we write $F(\boldsymbol{V})$ and $f_i(\boldsymbol{V})$, leaving the dependence of F and f_i on $\boldsymbol{X}$ and $\boldsymbol{x}_i$, respectively, implicit.

These problems are often solved (e.g., for the examples mentioned above) using so-called first-order iterative optimization methods, i.e., methods that iteratively update a solution based on the gradient of the loss function F, which we denote by ∇F. One example of such a method is GD, the update rule of which is

$$\boldsymbol{V}^{(t+1)} = G\left(\boldsymbol{V}^{(t)} - \eta \nabla F\left(\boldsymbol{V}^{(t)}\right)\right), \tag{2}$$

where t is the iteration index, η the stepsize, and G a projection operator (possibly the identity operator).

In this work, we consider a distributed scenario in which the rows of $\boldsymbol{X}$ are stored over N worker nodes, such that each node stores an equal fraction of the rows. The workers are responsible for computing the subgradients $\nabla f_1, \ldots, \nabla f_n$ and the coordinator is responsible for aggregating those subgradients and performing a gradient step.

Experimental setup

The results presented in this work are from experiments conducted on compute clusters hosted on Microsoft Azure (region `West Europe`), AWS (region `eu-north-1`), and the eX3 cluster.[1] For Azure, the nodes are of type `F2s_v2` and for AWS the nodes are of type `c5.xlarge`.[2] For AWS, we also provide traces for nodes of type `c5.xlarge` in region `us-east-1` and of type `t3.xlarge` in region `eu-north-1` [8]. The nodes used on eX3 are equipped with AMD EPYC 7302P processors and high-speed InfiniBand interconnects. We use the same type of node for the coordinator and the workers. Our implementation is written in the Julia programming language, and we use OpenMPI for communication—specifically, the `Isend` and `Irecv` nonblocking, point-to-point communication subroutines. For Azure and AWS, we use the CycleCloud and ParallelCluster systems, respectively, to create workers on-demand.

Throughout this paper, we consider a data matrix derived from the 1000 Genomes phase-3 dataset [39]. More precisely, we consider a binary representation of the data for each chromosome, where a nonzero entry in the (i, j)-th position indicates that the genome of the i-th subject differs from that of the reference genome in the j-th position. The matrix we use is the concatenation of such matrices computed for each chromosome. It is a sparse matrix of size $81\,271\,767 \times 2504$ with density about 5.360%. In Section 7, we also consider the HIGGS dataset, which consists of $11\,000\,000$ samples with 28 features [40]. For all computations, each worker stores the subset of the dataset assigned to it in memory throughout the computation.

3 Modeling the latency of gradient computations

In this section, we propose a model of the communication and computation latency of workers performing gradient computations in a distributed setting. Later, we use this model to predict the latency of the w-th fastest worker out of a set of workers. We first consider the latency of workers operating in *steady state* (Section 3.1), after which we consider how the latency of a particular worker changes over time (Section 3.2).

The model we propose is based on latency traces collected in clusters composed of up to 108 workers on AWS, Azure, and eX3, with varying per-worker computational load, which we denote by c, and b bytes communicated per iteration. Here, the computational load can be any quantity that captures the amount of work performed by each worker and iteration, such that a change in c results in a proportional change in the expected computation latency of a single worker. The number of bytes communicated and the computational load are equal for all workers, and we repeat the experiment at different days and times of the day.

In particular, for each worker, we record the latency associated with sending to the worker an iterate $\boldsymbol{V}$ and for the worker to respond with the result of the computation,

$$\boldsymbol{X}_{i:j}^{\mathsf{T}} \boldsymbol{X}_{i:j} \boldsymbol{V}, \tag{3}$$

for some integers $1 \leq i \leq j \leq n$, where $\boldsymbol{X}_{i:j}$ denotes the submatrix of $\boldsymbol{X}$ consisting of rows i through j. Hence, our results generalize to computations that rely on matrix multiplication, although the model is also easily adapted to other types of computations. In addition, we make available traces recorded for other computations and datasets, and we find consistent behavior across the computations and datasets considered [8].

We let c be the number of operations required to perform this computation, i.e., $c = 2\zeta dk(j - i + 1)$, where d is the dimension, k is the number of columns of $\boldsymbol{V}$, and ζ is the density of the data matrix. For all recordings, we randomly permute the rows of the matrix to break up dense blocks, and we adjust the computational load by tuning the number of samples processed.

In Fig. 1, we plot the range of computational loads considered, together with the mean and variance of the computation latency recorded for 100 different workers for each computational load, when the number of

1. See ex3.simula.no.
2. Both `F2s_v2` and `c5.xlarge` nodes are based on Intel Xeon Platinum 8000 series processors. `F2s_v2` nodes have an expected network speed of 875 Mbps, whereas `c5.xlarge` nodes have a network speed of up to 10 Gpbs.

bytes communicated per iteration is $b = 30\,048$. For reference, we also plot a line passing through the origin fitted to the data.

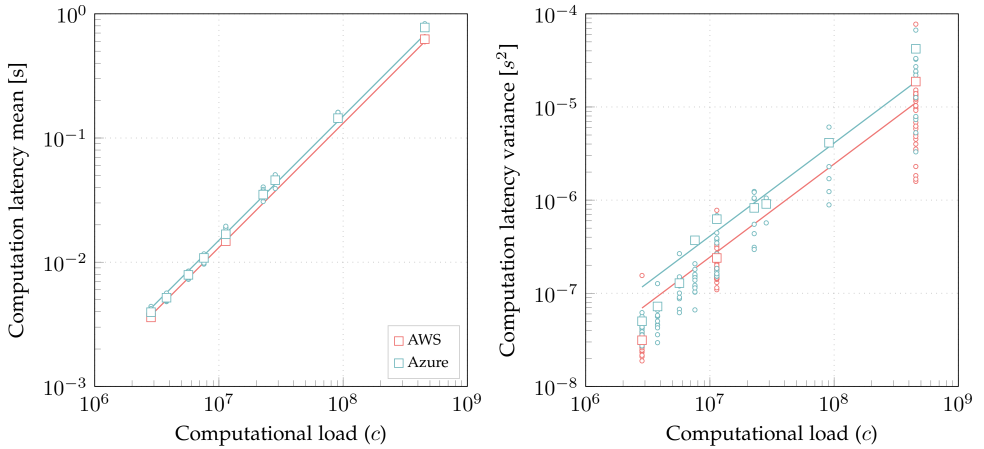


Fig. 1: Mean and variance of the computation latency recorded for 100 different workers as a function of the computational load. Circles correspond to workers, and the mean over all recordings for each computational load is marked by a square. For reference, we also show a line passing through the origin fitted to the data.

3.1 Steady-state latency

We find that the latency distribution of workers may change significantly over time, but that these changes typically occur quickly and that the distribution remains approximately constant between changes. Here, we characterize the latency of individual workers while in steady state. Our results are based on traces collected from running many iterations of (3) in sequence over a set of workers. For each iteration, we wait for all workers to return their result before proceeding to the next iteration. For this section, we have deliberately chosen traces for which the latency distribution does not change significantly throughout the computation.

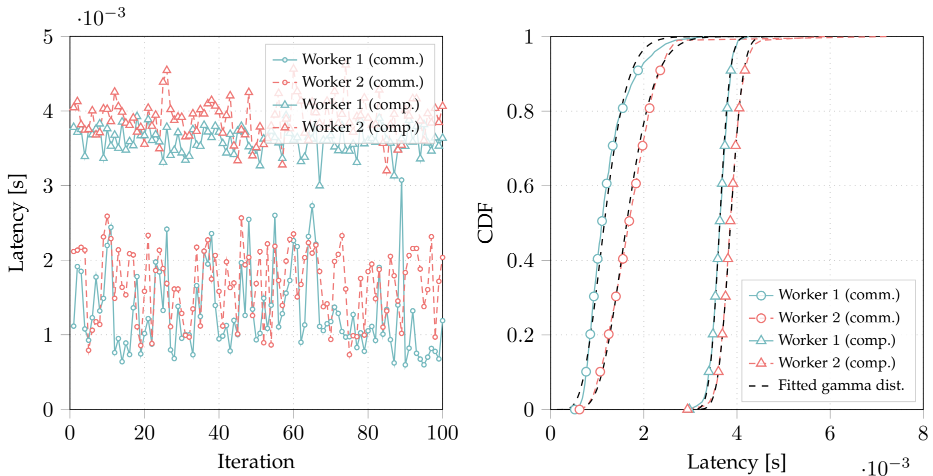


Fig. 2: Per-iteration latency (left) and empirical CDF of the per-iteration latency (right) of two workers on Azure, with $b = 30\,048$ bytes communicated per iteration (circles) and computational load $c = 2.841 \cdot 10^6$ (triangles). Worker 2 is, on average, 14% slower than worker 1. Black dashed lines indicate fitted gamma distributions.

In Fig. 2, we plot the communication latency (circles) and computation latency (triangles) recorded for two workers over 100 iterations (out of a total of 1600) on Azure. Note that the average latency differs between the two workers; worker 2 is about 14% slower. We show the associated cumulative distribution functions (CDFs) in Fig. 2. Now, for a set of workers, we model the latency of the i-th worker by the random variable

$$X_i^{(b,c)} = Y_i^{(b)} + Z_i^{(c)}, \tag{4}$$

where $Y_i^{(b)}$ and $Z_i^{(c)}$ are random variables associated with the communication and computation latency, respectively, of the worker, when the number of bytes communicated is b and the computational load is c.[3] We often omit the superscripts b and c.

We find that the communication and computation latency of workers on Azure and AWS is well-approximated by independent gamma-distributed random variables,[4] but that the parameters of these distributions typically differ between workers, i.e., probability distributions have to be fitted to the particular set of workers used for each computation, especially for systems like Azure CycleCloud and AWS ParallelCluster, which create new worker instances on-demand at the start of a computation. Failing to account for these differences can significantly reduce the accuracy of predictions made using the model; see Section 4.1 and Fig. 4.

3.2 Variability over time

The latency distribution of any particular worker typically changes over time. In particular, as a consequence of the design of cloud computing systems, where multiple virtual machines share the same physical host machine, workers experience *bursts* of higher latency. For example, performing memory-intensive operations, such as matrix multiplication, can more than halve the bandwidth available to other threads on the same machine [41].[5] Further, computations managed by cluster schedulers, such as Borg or Kubernetes, are often only guaranteed a very low fraction of the CPU cycles of the server it is assigned to, but may opportunistically use any cycles not used by other computations [42], [43, Ch. 14.3], potentially resulting in large performance fluctuations.

In Fig. 3, we show an example of such high-latency bursts, with the average latency of each of 3 workers out of the $N = 36$ workers used for a particular computation on AWS increasing by about 12% for about one minute.[6] The entire computation lasts for about 30 minutes, and most of the 36 workers experience at least one such burst over this time. Further, at least one worker is currently experiencing a burst of high latency for about 40% of the iterations. This problem becomes more severe for a larger number of workers—for computations consisting of hundreds of workers, the probability that no worker is currently experiencing a latency burst is close to zero.

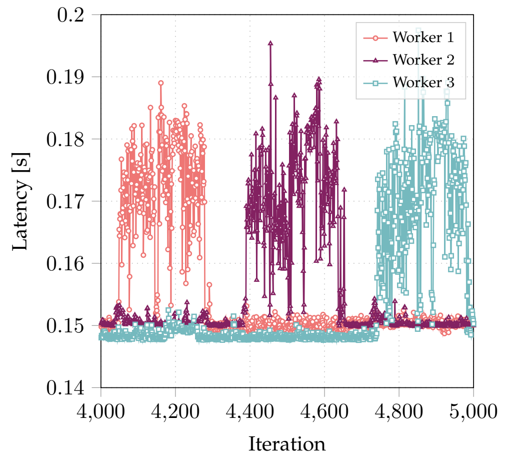


Fig. 3: Per-iteration computation latency of 3 workers (out of $N = 36$) on AWS, with computational load $c = 7.566 \cdot 10^7$. Workers typically experience bursts of high latency.

4 Predicting the Latency of Distributed Gradient Computations

Here, we show how to efficiently estimate the latency of the w-th fastest worker ($w \leq N$) of a set of workers, i.e., the w-th *order statistic* of the per-worker latency. Later, we use these predictions for dynamic load-balancing to minimize latency variations between workers (see Section 6). Throughout this section, we have deliberately chosen traces where workers are operating in steady state. When used for load-balancing, we account for bursts by dynamically updating the estimate of the latency distribution associated with each worker. We first consider the case where all workers are available at the start of each iteration (Section 4.1), before considering the more realistic case where workers may remain unavailable over several iterations (Section 4.2).

4.1 Order statistics latency

From the distributions of Y_i and Z_i for each worker, we can compute the distribution of the latency of the w-th fastest worker. However, the computational complexity of doing so analytically may be prohibitively high when the number of workers is large. Instead, we use Monte Carlo integration. The complexity of sampling from the latency of the w-th fastest worker is linear in the number of workers, since we can first

3. The model proposed in [19], where the latency of each worker is assumed to take on one of two discrete values, is similar to ours in the sense that latency may differ between workers. However, for our model, latency takes on values according to a continuous probability distribution.

4. In several previous works, latency is modeled by shifted exponential-distributed random variables. These models are related, since the sum of several exponential random variables is gamma-distributed. Hence, a possible interpretation is that the latency we record is the sum of the latency of several smaller computations, each of which has exponentially distributed latency.

5. This is known as the *noisy neighbor* problem.

6. Similar behavior was observed on AWS in [37], [38].

sample from Y_i and Z_i for each worker and then find the w-th smallest value of the resulting list in linear time, e.g., using the Quickselect algorithm. Through this process we can estimate, e.g., the expected latency of the w-th fastest worker.

In Fig. 4, we plot the average latency of the w-th fastest worker out of $N = 72$ workers for a particular computation with $b = 30048$ and $c = 4.545 \cdot 10^8$ on Azure. We also plot predictions made using Monte Carlo integration as explained above, and, for reference, predictions made by the commonly adopted i.i.d. model, where the latency of each worker is modeled by a random variable with mean and variance equal to the global mean and variance computed across all workers.[7] The proposed model yields an accurate prediction of the empirical performance, while assuming that latency is i.i.d. between workers can significantly reduce accuracy.

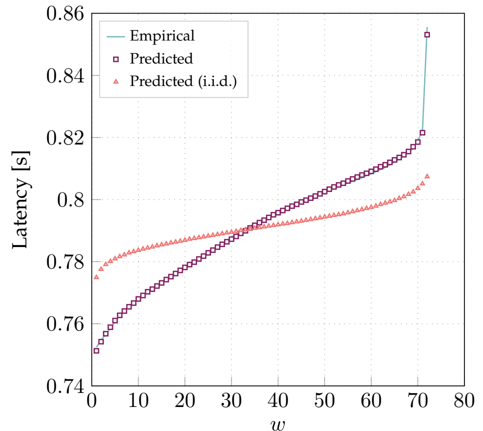


Fig. 4: Average latency of the w-th fastest worker out of $N = 72$ workers for a computation on Azure, and predicted latency, where the per-worker latency is modeled as either independent, but not necessarily identically distributed, or i.i.d., between workers. The i.i.d. assumption can significantly reduce accuracy.

4.2 Order statistics latency of iterative computations

In Section 4.1, we considered order statistics in cases where all workers are available at the start of each iteration. However, for straggler-resilient methods, we wish to proceed to the next iteration immediately after receiving results from the w fastest workers, without waiting for the remaining $N - w$ workers, which may remain unavailable over several subsequent iterations.

Here, we show how to estimate latency in this scenario. Denote by $T_w^{(t)}$ the time at which the t-th iteration of an iterative computation, for which the coordinator waits for the w-th fastest worker in each iteration, is completed (i.e., the latency of the t-th iteration is $T_w^{(t)} - T_w^{(t-1)}$). We wish to simulate the time series process $T_w^{(1)}, \dots, T_w^{(\ell)}$, where ℓ is the number of iterations. We do so by using a two-state model, where workers are either idle or busy. First, each worker has a local first-in-last-out task queue of length 1. If the i-th worker is idle and there is a task in its queue, it immediately removes the task from the queue and becomes busy for a random amount of time, which is captured by the random variable X_i (recall that we can sample from X_i, see Section 3.1), before becoming idle again. At the start of each iteration, the coordinator assigns a task to each worker, and once w of those tasks have been completed, the coordinator proceeds to the next iteration.

Using this model, we can efficiently simulate realizations of $T_w^{(1)}, \dots, T_w^{(\ell)}$ by using a priority queue data structure (see, e.g., [44]) to map the index of each worker to the next time at which it will transition from busy to idle. This strategy is typically referred to as *event-driven* simulation. By performing such simulations we can estimate, e.g., the expected time required to perform ℓ iterations, in a manner that accounts for the fact that workers may remain unavailable over several iterations. We provide an implementation of such a simulator in [8].

In Fig. 5, we plot the cumulative latency over 100 iterations for two jobs, with $b = 30\,048$, $c = 7.575 \cdot 10^6$, and $N = 72$ on AWS, where, in one job, we wait for $w = 9$ workers (blue curves) and, in the other, for all $w = N = 72$ workers (red curves). We also plot the predictions made by the proposed model based on event-driven simulations, which accounts for the interaction between iterations, and the model described in Section 4.1, which does not. For $w = N = 72$, both models give accurate predictions. However, for $w = 9 < N$, the model of Section 4.1 underestimates the overall latency, since it does not account for the case where workers remain unavailable over multiple iterations. The model based on event-driven simulations remains accurate.

5 DSAG

In this section, we consider learning in cloud computing systems. In particular, we want an optimization method that i) is able to make progress even when some workers fail to respond, ii) has fast initial convergence, similar to SGD, which is achieved by performing many fast, but inexact, iterations, iii) eventually converges to the optimum, iv) allows for dynamic load-balancing, and v) has low update complexity. GD and SAG fail points i) and iv), SGD fails point iii), and coded computing methods fail either point ii) or iii), and, in most cases, points iv) and v).

7. We model the latency distribution by a gamma distribution, which we find provides more accurate predictions than the more commonly used shifted exponential distribution.

To address i)–v), we introduce DSAG, which adapts SAG to distributed environments with heterogeneous and straggling workers. As with SAG, the key idea is to cache stale subgradients. However, unlike SAG, DSAG utilizes subgradients computed in previous iterations that arrive late. Further, DSAG allows for load-balancing by dynamically changing the number of data partitions (and hence the number of samples that make up each partition). DSAG meets all of the above criteria.

DSAG works as follows. Denote by

$$\boldsymbol{Y}_{i:j}^{(t)} \triangleq \sum_{k=i}^{j} \nabla f_k \left(\boldsymbol{V}^{(t)} \right)$$

the subgradient computed from samples i through j, where $j \geq i$. The coordinator maintains a set of such subgradients, denoted by $\mathcal{Y}$, which we refer to as the gradient cache. Upon receiving a subgradient $\boldsymbol{Y}_{i:j}^{(t)}$ from a worker, the coordinator first selects the subset of overlapping subgradients

$$\mathcal{Y}' \triangleq \left\{ \boldsymbol{Y}_{i':j'}^{(t')} \in \mathcal{Y} \ : \ i \leq i' \leq j \text{ or } i \leq j' \leq j \right\}.$$

If any such subgradient is more recent than the received subgradient (i.e., if $t' \geq t$ for some $\boldsymbol{Y}_{i':j'}^{(t')} \in \mathcal{Y}'$), the process is aborted and the received subgradient discarded. Otherwise, the overlapping subgradients are discarded in favour of the received subgradient, i.e.,

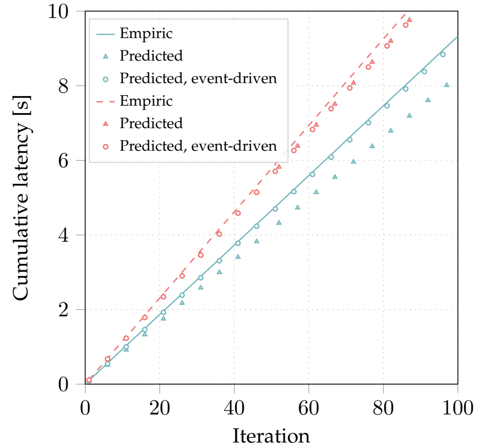


Fig. 5: Cumulative latency over 100 iterations. Blue curves correspond to $w = 9$ and red curves to $w = 72$. Each iteration ends once w workers have completed their task. For $w < N$, we need to account for the case where workers remain unavailable over several iterations, which we do using the model based on event-driven simulations.

$$\mathcal{Y} \leftarrow (\mathcal{Y} \setminus \mathcal{Y}') \cup \left\{ \boldsymbol{Y}_{i:j}^{(t)} \right\}.$$

This process allows for changing partition boundaries at runtime, e.g., due to load-balancing, and can be implemented efficiently by storing the elements of $\mathcal{Y}$ as nodes in a tree data structure.[8] Denote by

$$\boldsymbol{H} \triangleq \sum_{y \in \mathcal{Y}} y \tag{5}$$

the sum of the elements of $\mathcal{Y}$. The coordinator maintains this sum by assigning

$$\boldsymbol{H} \leftarrow \boldsymbol{H} + \boldsymbol{Y}_{i:j}^{(t)} - \sum_{y \in \mathcal{Y}'} y$$

whenever a subgradient $\boldsymbol{Y}_{i:j}^{(t)}$ is inserted into $\mathcal{Y}$. Finally, $\boldsymbol{H}$ is used in place of the exact gradient ∇F to update $\boldsymbol{V}^{(t)}$, i.e.,

$$\boldsymbol{V}^{(t+1)} = G\left(\boldsymbol{V}^{(t)} - \eta \left(\frac{1}{\xi} \boldsymbol{H} + \nabla R \left(\boldsymbol{V}^{(t)} \right) \right) \right), \tag{6}$$

where ξ is the fraction of samples covered by the elements of $\mathcal{Y}$. Scaling the gradient in this way improves the rate of convergence for the iterations before the coordinator has received subgradients covering all samples.[9]

We remark that if there exists $\boldsymbol{Y}_{i':j'}^{(t')}$ in $\mathcal{Y}$ such that $i' = i$ and $j' = j$, the existing element can be updated in-place. In this case, and if the received subgradient is computed from the most recent iterate, the update process degrades to that of SAG.

5.1 Distributed implementation

Here, we describe our distributed implementation of DSAG. In particular, we wish to maintain predictable and low latency in the presence of stragglers. For SAG or SGD, this can be achieved by only waiting for a subset of workers to return in each iteration, and ignoring any results computed by straggling workers. However, since the same workers are likely to be stragglers for extended periods of time, the subgradients received from the fastest subset of workers by the coordinator will not be selected uniformly at random, unless all workers store the entire dataset or the coordinator waits for all workers. This can significantly reduce the rate of convergence, since parts of the dataset may never factor into the learning process (see Section 7 and Fig. 7).

8. When using a tree data structure, the complexity of deleting and inserting subgradients is in $\mathcal{O}(\log |\mathcal{Y}|)$.
9. A similar scaling is used by SAG.

DSAG addresses this shortcoming by utilizing stale results and through dynamic load-balancing. In particular, at the t-th iteration, the coordinator waits until it has received subgradients computed from $\boldsymbol{V}^{(t)}$ from at least w workers. During this time, the coordinator may also have received subgradients from previous iterations, which the coordinator stores if they are less stale than the currently stored subgradients it would replace. Further, we allow for a small margin, such that after receiving the w-th fresh subgradient, the coordinator waits for 2% longer—collecting any subgradients received during this time—before updating the iterate. We find that doing so can improve the rate of convergence at the expense of a small increase in latency, especially when combined with load-balancing. We explain the load-balancing strategy that we propose in Section 6.

5.2 Convergence of DSAG

DSAG builds upon the SAG method, for which the error

$$F\left(\boldsymbol{V}^{(t)}\right) - F\left(\boldsymbol{V}^{*}\right),$$

where $\boldsymbol{V}^{*}$ is the optimum, decreases with $\mathcal{O}(1/t)$ and $\mathcal{O}(\rho^t)$, for some $\rho < 1$, for convex and strongly convex problems, respectively [7]. We do not have convergence proofs for DSAG—the analysis of asynchronous optimization methods is notoriously challenging, and the analysis of SAG is already complex—but we make a few remarks to relate the behavior of DSAG to that of SAG.

SAG updates one subgradient, selected uniformly at random over all partitions, at each iteration, and does not make use of stale subgradients. DSAG differs by updating one or more subgradients per iteration, and in that some of the updated subgradients may have been computed from a previous iterate, provided they are less stale than the replaced subgradients. Hence, the subgradients utilized by DSAG are at least as fresh as those used by SAG. Second, DSAG, unlike SAG, may discard cached subgradients if it receives a subgradient that is not aligned with an already cached subgradient (SAG does not support changing the partition boundaries at runtime).

Hence, we conjecture that the rate of convergence of DSAG is at least as good as that of SAG for iterations when no subgradients are discarded, and that it is worse than that of SAG in iterations where cached subgradients have been discarded, and until the discarded entries have been repopulated. We present empirical results that support this conjecture, see Section 7.

6 Load-Balancing

Recall that computing speed typically differs between workers and may change over time (see Section 3). Unless these differences are accounted for, fast workers typically spend a significant amount of time waiting for slower ones, and some workers may never be among the w fastest ones. Here, we propose a strategy to dynamically adjust the size of the data partitions stored by each worker to alleviate this issue. The process consists of three steps:

1) Latency profiling to estimate the probability distribution of Z_i and Y_i for each worker based on recorded latency (see Section 6.1).
2) Optimizing the number of subpartitions for each worker using simulations based on the latency model of Section 4.2 to predict the impact of each change (see Section 6.2).
3) Re-partitioning the local dataset for any workers for which the number of subpartitions has changed (see Section 6.3).

All three steps are performed asynchronously in parallel and are running continuously in the background. In particular, whenever the optimizer finishes, it is restarted to include any new latency recordings.[10] We show how load-balancing affects latency in Fig. 6, and we describe the three steps in detail next.

10. The load-balancer proposed in [37] takes a similar approach, but is designed for web services.

6.1 Latency profiler

The latency profiler is responsible for estimating the mean and variance of the communication and computation latency of each worker, and for providing these to the optimizer. It takes as its input latency recorded both by the coordinator and the workers themselves. In particular, for each worker, the coordinator records the time between sending an iterate to the worker and receiving a response. Meanwhile, the workers record the time between starting to process a received iterate and having a response ready, and include this recording in their responses.

For each worker, we take the latency recorded by the worker as a sample of the computation latency, and the difference between the latency recorded by the worker and coordinator as a sample of the communication latency, i.e., for the i-th worker, as realizations of Z_i and Y_i, respectively. Hence, we record the round-trip communication latency, which includes the time required for data to be sent over the wire and any queuing at either end.

Next, for each worker, the profiler computes the sample mean and variance over a moving time window, i.e., samples older than a given deadline (in seconds) are discarded before processing. Choosing a window size involves making a trade-off—a larger window size makes statistics computed over it less noisy, but increases the time needed for the profiler to adapt to changes.[11] We denote by $e_{\mathsf{Y},i}$ and $v_{\mathsf{Y},i}$ the mean and variance of the communication latency of the i-th worker, and by $e_{\mathsf{Z},i}$ and $v_{\mathsf{Z},i}$ the mean and variance of the computation latency, computed as described above. For each worker, whenever new latency recordings are available, the mean and variance of its communication and computation latency are re-computed and sent to the optimizer, which uses them to fit probability distributions.[12]

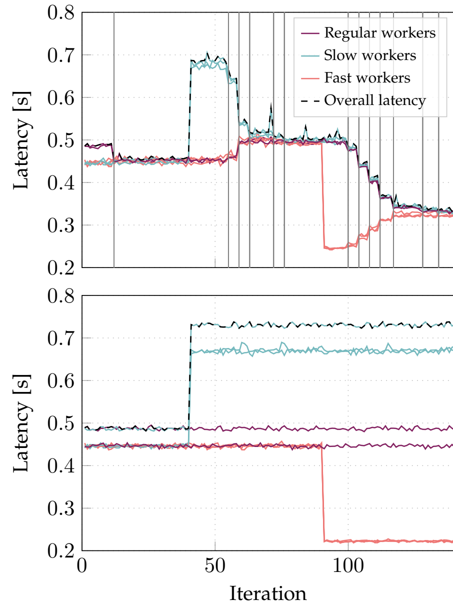


Fig. 6: Per-worker latency for $N = 8$ workers with (top) and without (bottom) load-balancing, when waiting for all workers (i.e., $w = N$). We artificially slow down 3 randomly selected workers (blue lines) after 40 iterations, and speed up another set of 3 randomly selected workers (red lines) after 90 iterations. Note that there is some natural variation in addition. The load-balancer automatically re-balances the workload in the iterations marked with gray lines. For the final 20 iterations, the overall latency of the unbalanced system is more than twice that of the system with load-balancing.

6.2 Optimizer

For each worker, we tune its workload by changing the number of subpartitions that the data it stores locally is divided into. The optimizer takes as its input the most recent statistics computed by the profiler and a vector $\boldsymbol{p} = [p_1, \ldots, p_N]$ containing the current number of subpartitions for each worker, and returns an updated vector $\boldsymbol{p}' = [p'_1, \ldots, p'_N]$. For any solution $\boldsymbol{p}$, we impose a constraint on the expected overall per-iteration *contribution*, which we define as

$$h(\boldsymbol{p}) \triangleq \sum_{i=1}^{N} h_i(\boldsymbol{p}), \text{ with } h_i(\boldsymbol{p}) \triangleq \frac{u_i(\boldsymbol{p}) n_i}{p_i n},$$

where n_i is the number of samples stored by the i-th worker and $u_i(\boldsymbol{p})$ the fraction of iterations that the i-th worker delivers a fresh result in. Hence, $h_i(\boldsymbol{p})$ is a measure of the extent to which the i-th worker contributes to the learning process. Note that u_i is a nonlinear function of $\boldsymbol{p}$, i.e., it depends on the workload of the entire set of workers. The goal of the optimizer is to minimize latency variation between workers within this constraint. More formally, its goal is to solve

$$\begin{aligned} \arg\min_{\boldsymbol{p}'} \quad & \frac{\max\left\{e'_{\mathsf{X},1}, \ldots, e'_{\mathsf{X},N}\right\}}{\min\left\{e'_{\mathsf{X},1}, \ldots, e'_{\mathsf{X},N}\right\}} \\ \text{s.t.} \quad & h(\boldsymbol{p}') \geq h_{\mathsf{min}}, \end{aligned} \tag{7}$$

11. We use a window size of 10 seconds, which we find is a good trade-off for the applications we consider.

12. The shape and scale parameter of a gamma-distributed random variable with mean e and variance v is e^2/v and v/e, respectively.

Algorithm 1 Load-balancer

```
procedure OPTIMIZE(p)
    p' ← p
    i ← arg max [e'_{X,1}, ..., e'_{X,N}]                      ▷ Slowest worker
    for j = 1, ..., N do
        p'_j ← ⌊ e_{Z,j} p_j / (e_{Y,i} + e_{Z,i} − e_{Y,j}) ⌋  ▷ Equalize total latency
    end for
    while h(p') < h_min do
        i ← arg min [e'_{X,1}, ..., e'_{X,N}]                  ▷ Fastest worker
        p'_i ← ⌊0.99 · p'_i⌋                                   ▷ Increase workload
    end while
    while h(p') ≥ 0.99 · h_min do
        i ← arg max [e'_{X,1}, ..., e'_{X,N}]                  ▷ Slowest worker
        p'_i ← ⌈1.01 · p'_i⌉                                   ▷ Decrease workload
    end while
    return p'
end procedure
loop                                                           ▷ Optimizer main loop
    Collect updated latency statistics from the profiler
    p ← OPTIMIZE(p)
    Distribute the updated vector p
end loop
```

where h_{min} is the constraint, and $e'_{\mathsf{X},i}$ is the expected overall latency of the i-th worker if its local dataset is split into p'_i subpartitions. Throughout the optimization process, we use the approximations

$$e'_{\mathsf{Z},i} \triangleq e_{\mathsf{Z},i}\frac{p_i}{p'_i}\ , \quad v'_{\mathsf{Z},i} \triangleq v_{\mathsf{Z},i}\frac{p_i^2}{{p'_i}^2},$$

and

$$e'_{\mathsf{X},i} \triangleq e_{\mathsf{Y},i} + e'_{\mathsf{Z},i},$$

where p_i is the current number of subpartitions of the i-th worker. Hence, we linearize the mean and variance of the computation latency around the value of p_i for which it was recorded.[13]

It is difficult to compute u_i, and thus h, analytically, but u_i can be estimated via event-driven simulations as explained in Section 4.2.[14] However, this requires that the optimizer i) is robust against noise in the estimates of u_i, and ii) evaluates u_i a small enough number of times to be computationally fast enough to provide useful solutions in time. We find that traditional optimization techniques that, e.g., rely on gradients, fail the first criteria, while *meta-heuristic* techniques (e.g., evolutionary algorithms) fail the second. Hence, we propose an optimizer that solves (7) by making small changes to $\boldsymbol{p}$ in an iterative fashion.

At a high level, the optimizer attempts to increase the contribution of workers that are always among the w fastest by giving them more work, without increasing the overall latency. This increases the overall per-iteration contribution, thus giving the optimizer leeway to reduce the overall iteration latency by reducing the workload of the slowest workers. The proposed algorithm is given in Algorithm 1. Since h is estimated via simulations, we evaluate the constraint with a 1% tolerance. Finally, we set the constraint to be

$$h_{\mathsf{min}} = h(\boldsymbol{p}_0),$$

where $\boldsymbol{p}_0$ is the baseline number of subpartitions for each worker used at the start of the first iteration. This is to ensure that load-balancing does not reduce the rate of convergence.

6.3 Re-partitioning

Whenever the optimizer produces an updated number of subpartitions for a particular worker, the update is included with the next iterate sent to the worker, which re-partitions its local dataset. However, re-partitioning carries a cost, since it invalidates subgradients cached by the coordinator. Here, we show how to minimize the number and impact of such *cache evictions* resulting from re-partitioning. First, we partition the data matrix such that the i-th worker stores locally the submatrix

$$\boldsymbol{X}^{(i)} \triangleq \boldsymbol{X}_{p_{\mathsf{start}}(n,N,i):p_{\mathsf{stop}}(n,N,i)},$$

13. This linearization is motivated by Fig. 1. If latency has been measured for several different values of p_i, we use a weighted average over the values of p_i for which we have recordings.

14. With our implementation, for $N = 100$ workers and $w = 50$, simulating 100 iterations of the learning process takes about 1.5 milliseconds.

Algorithm 2 Partition alignment

1: $k_i \leftarrow \mathrm{mod}\,(k_i, p_i) + 1$
2: $k'_i \leftarrow p_{\mathsf{trans}}(n_i, p_i, p'_i, k_i)$
3: **while** $p_{\mathsf{start}}(n_i, p'_i, k'_i) \neq p_{\mathsf{start}}(n_i, p_i, k_i)$ **do**
4: $\quad k'_i \leftarrow k'_i - 1$
5: $\quad k_i \leftarrow p_{\mathsf{trans}}(n, p'_i, p_i, k'_i)$
6: **end while**
7: $p_i \leftarrow p'_i$
8: $k_i \leftarrow k'_i$

where

$$p_{\mathsf{start}}(n, p, i) = \left\lfloor \frac{(i-1)\,n}{p} \right\rfloor + 1$$

and

$$p_{\mathsf{stop}}(n, p, i) = \left\lfloor \frac{in}{p} \right\rfloor,$$

with $1 \leq p \leq n$ and $1 \leq i \leq p$. Next, for each worker, we subpartition the data it stores locally, such that, in each iteration, the i-th worker processes the matrix

$$\boldsymbol{X}^{(i)}_{p_{\mathsf{start}}(n_i, p_i, k_i):p_{\mathsf{stop}}(n_i, p_i, k_i)},$$

for some index k_i. Hence, we may tune the workload of a worker by sending it a new value p_i, which changes the number of samples processed per iteration. The following example shows how doing so leads to cache evictions.

Example 1 (Re-partitioning)**.** *Consider a scenario with* 2 *workers,* $n_1 = n_2 = 10$ *(i.e.,* $n = 20$*), and* $p_1 = p_2 = 2$*, such that the partitions on the first worker are* $\boldsymbol{X}_{1:5}$ *and* $\boldsymbol{X}_{6:10}$*, and* $\boldsymbol{X}_{11:15}$ *and* $\boldsymbol{X}_{16:20}$ *on the second. Now, say that we let* $p_1 \leftarrow 3$*, such that the partitions on the first worker are* $\boldsymbol{X}_{1:3}$*,* $\boldsymbol{X}_{4:6}$*, and* $\boldsymbol{X}_{7:10}$*. Prior to this change, the coordinator stores gradients corresponding to partitions* $\boldsymbol{X}_{1:5}$ *and* $\boldsymbol{X}_{6:10}$*. Now, if in the next iteration the worker sends to the coordinator the subgradient computed over* $\boldsymbol{X}_{4:6}$*, both of the existing entries need to be evicted before inserting the new subgradient, leading to a lower rate of convergence until the missing cache entries have been populated.*

We find that cache evictions due to re-partitioning can significantly reduce the rate of convergence, since the gradient used by DSAG no longer covers all samples of the dataset. We use two strategies to reduce the severity of this issue. First, we refrain from distributing an update $\boldsymbol{p}'$ to the workers until doing so would improve the objective function (7) by more than some threshold (e.g., 10%). Second, we process subpartitions in order to minimize the number of iterations for which evicted cache entries remain empty. More formally, the i-th worker stores a counter k_i that it increments in a cyclic fashion each time it receives an iterate, i.e.,[15]

$$k_i \leftarrow \mathrm{mod}\,(k_i, p_i) + 1. \tag{8}$$

Next, it computes the gradient with respect to the k_i-th of its locally stored partitions. We show the benefit of this approach with the following example.

Example 2 (Continuation of Example 1)**.** *Immediately after re-partitioning, the coordinator stores subgradients computed over partitions* $\boldsymbol{X}_{1:5}$ *and* $\boldsymbol{X}_{6:10}$ *(we omit partitions stored by the second worker). To minimize cache evictions, over the following* 3 *iterations, the first worker sends to the coordinator:*

1) *The gradient over* $\boldsymbol{X}_{1:3}$*, evicting the gradient over* $\boldsymbol{X}_{1:5}$*, resulting in a cache with the gradients over* $\boldsymbol{X}_{1:3}$ *and* $\boldsymbol{X}_{6:10}$*, leaving the gradient over* $\boldsymbol{X}_{4:5}$ *missing.*
2) *The gradient over* $\boldsymbol{X}_{4:6}$ *, evicting the gradient over* $\boldsymbol{X}_{6:10}$*, resulting in a cache with the gradients over* $\boldsymbol{X}_{1:3}$ *and* $\boldsymbol{X}_{4:6}$*, leaving the gradient over* $\boldsymbol{X}_{7:10}$ *missing.*
3) *The gradient over* $\boldsymbol{X}_{7:10}$*, resulting in a cache with the gradients over* $\boldsymbol{X}_{1:3}$*,* $\boldsymbol{X}_{4:6}$*, and* $\boldsymbol{X}_{7:10}$*, leaving no missing entries.*

In this case, the gradients over $\boldsymbol{X}_{4:5}$ *and* $\boldsymbol{X}_{7:10}$ *are missing from the cache for* 1 *iteration each. If instead the worker had started by sending the gradient over* $\boldsymbol{X}_{4:6}$*, either the gradient over* $\boldsymbol{X}_{1:3}$ *or* $\boldsymbol{X}_{7:10}$ *would have been missing for* 2 *iterations, and the other for* 1 *iteration, resulting in a lower rate of convergence.*

This approach is most effective if the first sample of the partition processed immediately after a re-partitioning is aligned with the first sample of a partition already in the cache, since otherwise the evicted entries are not re-populated until after a full pass over the data (this happens if the first worker in Example 2 starts by processing $\boldsymbol{X}_{4:6}$ after re-partitioning). Hence, when changing the number of subpartitions of the i-th worker from p_i to p'_i, instead of using (8), we update k_i using Algorithm 2, which relies on the function

$$p_{\mathsf{trans}}(n_i, p_i, p'_i, k_i) = \left\lceil p_{\mathsf{start}}(n_i, p_i, k_i)\frac{p'_i}{n_i} \right\rceil,$$

15. Note that, when $w < N$, workers, unlike the coordinator, are unaware of the current iteration index since they may have remained unavailable for an arbitrary amount of time.

that returns the index of the partition containing sample $p_{\text{start}}(n_i, p_i, k_i)$ when the number of partitions is p_i'. We illustrate Algorithm 2 with Example 3.

Example 3 (Continuation of Example 2)**.** *Say that, prior to re-partitioning, the first worker processed partition $\boldsymbol{X}_{1:5}$, so that $k_1 = 1$, and that we are changing the number of subpartitions from $p_1 = 2$ to $p_1' = 3$. In this case, the $n_1 = 10$ samples stored by the first worker are subpartitioned as follows,*

$$\begin{array}{llllllllllll} p_1 = 2: & [1, & 2, & 3\,, & 4, & 5], & [6\,, & 7, & 8, & 9, & 10] \\ p_1' = 3: & [1, & 2, & 3], & [4, & 5, & 6], & [7, & 8, & 9, & 10] \end{array}$$

where the indices are the row indices of $\boldsymbol{X}$, and brackets in the first and second line indicate partition boundaries before and after re-partitioning, respectively. Now, Algorithm 2 finds a partition out of $p_1' = 3$ partitions such that its first sample is equal to that of some partition out of $p_1 = 2$. It proceeds as follows. First, let $k_1 \leftarrow \operatorname{mod}(1,2) + 1 = 2$ (Algorithm 2), and $k_1' \leftarrow p_{\text{trans}}(10, 2, 3, k_1) = 2$ (Algorithm 2). Since the k_1-th and k_1'-th partitions are not aligned (Algorithm 2)—$p_{\text{start}}(10, 3, k_1') = 4 \neq 6 = p_{\text{start}}(10, 2, k_1)$—we let $k_1' \leftarrow k_1' - 1 = 1$ and $k_1 \leftarrow p_{\text{trans}}(10, 3, 2, k_1') = 1$ (Algorithm 2). Now the partitions are aligned (Algorithm 2)—$p_{\text{start}}(10, 2, k_1) = 1 = p_{\text{start}}(10, 3, k_1')$—and the worker assigns $p_1 \leftarrow p_1'$ and $k_1 \leftarrow k_1'$ (Algorithm 2).

Note that Algorithm 2 always terminates, since the first partition always starts at the first sample stored by the worker, i.e., $k_i = k_i' = 1$ results in the partitions being aligned regardless of the values of p_i and p_i'. However, $k_i = k_i' = 1$ may not be the only solution. For example, if $n_i = 10$, $p_i = 2$, and $p_i' = 4$, then $k_i = 2$ and $k_i' = 3$ also results in aligned partitions—$p_{\text{start}}(10, 4, 3) = 6 = p_{\text{start}}(10, 2, 2)$. Hence, Algorithm 2 improves timeliness, since always setting $k_i = k_i' = 1$ after re-partitioning could result in the first few subpartitions being processed much more frequently than the others.

7 Convergence results

Here, we evaluate the performance of DSAG for PCA and logistic regression, and compare it to that of GD, SGD, SAG, and coded computing methods, on eX3 and AWS (see Section 2 for details). We also evaluate the impact of load-balancing on performance for DSAG, SAG, and SGD. For PCA, the loss function is given by

$$R(\boldsymbol{V}) = \frac{1}{2}\|\boldsymbol{V}\|_{\mathsf{F}}^2 \quad \text{and} \quad f_i(\boldsymbol{V}) = \frac{1}{2}\left\|\boldsymbol{x}_i - \boldsymbol{x}_i\boldsymbol{V}\boldsymbol{V}^{\mathsf{T}}\right\|^2, \tag{9}$$

where the columns of $\boldsymbol{V}$ make up the computed principal components, $\|\cdot\|$ denotes the Euclidean norm, and $\|\cdot\|_{\mathsf{F}}$ denotes the Frobenius norm and $\boldsymbol{V}$ is updated according to (2). For PCA, $G(\cdot)$ in (2) is the Gram-Schmidt operator, i.e., $G(\cdot)$ takes an input matrix and applies the Gram-Schmidt orthogonalization procedure to its columns such that the columns of the resulting matrix form an orthonormal basis with the same span as the columns of the input matrix. For logistic regression, the loss is the *L2-regularized* classification error, i.e.,

$$R(\boldsymbol{V}) = \frac{\lambda}{2}\|\boldsymbol{V}\|^2 \quad \text{and} \quad f_i(\boldsymbol{V}) = \frac{\log\left[1 + \exp\left(-b_i \boldsymbol{x}_i^{\mathsf{T}}\boldsymbol{V}\right)\right]}{n},$$

where $b_1, \ldots, b_n$ are the classification labels, with $b_i \in \{-1, +1\}$, λ is the regularization coefficient, and in this case $G(\cdot)$ is the identity operator. For PCA, we use a matrix derived from the 1000 Genomes phase-3 dataset [39], and for logistic regression we use the HIGGS dataset [40] (see Section 2). For PCA, we compute the top 3 principle components, and for logistic regression, as in [7], we normalize all features to have zero mean and unit variance, add an intercept equal to 1, and set the regularization coefficient to 1 divided by the number of samples, i.e., $\lambda = 1/11\,000\,000$. We use 100 and 10 subpartitions for PCA and logistic regression, respectively.

We measure performance as the latency to solve either PCA or logistic regression to within some precision of the optimum, and, for all scenarios, we plot the suboptimality gap, i.e., the difference between the explained variance (for PCA) or classification error (for logistic regression) of the computed solution and that of the optimum, as a function of time. The results shown are averages over 5 experiments conducted on the respective computing systems. For GD and coded computing, we use a stepsize of $\eta = 1.0$ for both PCA and logistic regression, whereas for DSAG, SAG, and SGD, we use a stepsize of $\eta = 0.9$ for PCA and $\eta = 0.25$ for logistic regression (we need to reduce the stepsize relative to GD for the stochastic methods to ensure convergence). We remark that GD applied to solving the optimization problem in (1) with the loss function in (9) with $\eta = 1.0$ is equivalent to the *power method* for PCA, i.e., the power method is a special case of GD.

7.1 Coded computing

Coded computing methods with *code rate r* (a quantity between 0 and 1) make it possible to either recover the gradient exactly (e.g., [4]) or an approximation thereof (e.g., [5], [6], [21], [22]) from intermediate results computed by a subset of the workers, at the expense of increasing the computational load of each worker by a factor $1/r$ relative to GD. The gradient is recovered via a decoding operation (that typically reduces to solving a system of linear equations), the complexity of which usually increases superlinearly with the number of workers. Ideally, the gradient can be recovered exactly from the results computed by any set of

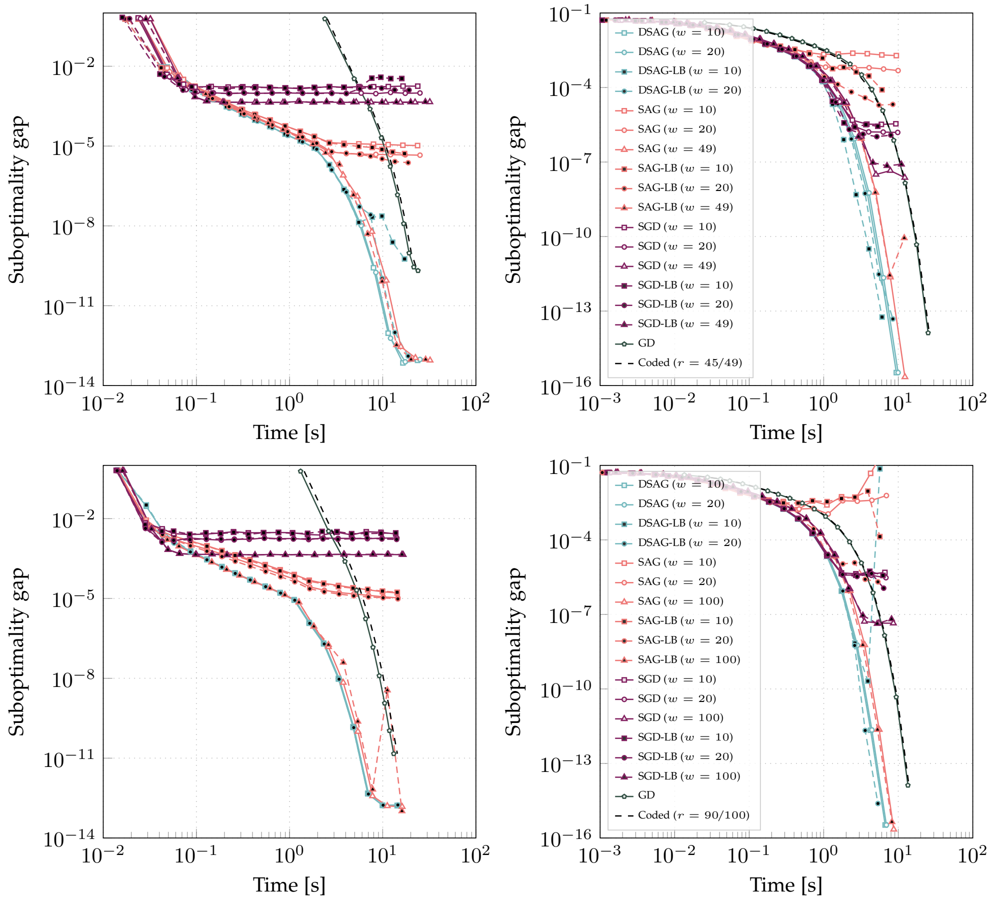


Fig. 7: Convergence of PCA (left column) and logistic regression (right column) for $N = 49$ workers on eX3 (top row) and $N = 100$ workers on AWS (bottom row). The dataset is split evenly over the workers and is initially subdivided into 100 subpartitions for PCA and 10 subpartitions for logistic regression. Stochastic optimization methods with $w < N$ effectively reduce the impact on latency of straggling workers, but only DSAG ensures convergence to the optimum. Load-balancing can improve latency further in some instances. The results shown are averages over 5 experiments.

$\lceil rN \rceil$ workers—codes with this property are referred to as *maximum distance separable* (MDS) codes—but increasing the number of results required can allow for reducing the decoding complexity [9].

To compare against the wide range of coded computing methods, we use an idealized estimate derived from the GD results. In particular, we assume that the code is MDS, but that the decoding complexity is zero. More specifically, we set the latency per iteration equal to that of the $\lceil rN \rceil$-th fastest worker after scaling the computational latency recorded for GD of all workers by $1/r$, and the rate of convergence equal to that of GD. Hence, both the latency and rate of convergence of the estimate are bounds on what is achievable with coded computing. Further, for PCA, this bound includes coded computing methods for matrix multiplication (e.g., [3], [6], [9], [10]), since GD is equivalent to the power method in this instance.

7.2 Artificial scenario

While we are primarily interested in cloud computing systems, for the sake of reproducibility, we first present results recorded for $N = 49$ workers on eX3, which is much more homogenous than the cloud, where we introduce variability in a controlled manner. In particular, we artificially increase the computational latency of the i-th worker by a factor $(i/N) \cdot 0.4$ by introducing delays at the worker nodes.[16] Further, we remove this artificial latency for workers 40 through 49 after one second has passed from the start of the learning process to simulate those workers coming out of a high-latency burst.

In Fig. 7 (top row), we show convergence of PCA (left) and logistic regression (right) in this scenario. First, for both PCA and logistic regression, at least one of the stochastic methods (DSAG, SAG, and SGD) is more

16. This level of variability is comparable to what we have observed for instances of type `F2s_v2` on Azure.

TABLE 1: Approximate latency of stochastic methods.

	Comm. latency [s]	Comp. latency [s]
eX3 PCA	$2.0 \cdot 10^{-5}$ to $6 \cdot 10^{-5}$	$2.2 \cdot 10^{-2}$ to $3.1 \cdot 10^{-2}$
AWS PCA	$1.5 \cdot 10^{-4}$ to $1 \cdot 10^{-3}$	$1.3 \cdot 10^{-2}$ to $1.6 \cdot 10^{-2}$
eX3 Logistic regression	$0.2 \cdot 10^{-5}$ to $3 \cdot 10^{-5}$	$1.8 \cdot 10^{-3}$ to $2.5 \cdot 10^{-3}$
AWS Logistic regression	$1.0 \cdot 10^{-4}$ to $6 \cdot 10^{-4}$	$1.1 \cdot 10^{-3}$ to $1.3 \cdot 10^{-3}$

than twice as fast as GD for any suboptimality gap—performing many fast, but inexact iterations, is often preferable to performing fewer more accurate iterations. However, for SAG, when $w < N$, and SGD, there is a point beyond which convergence effectively stops. For SGD, the high variance of its gradient estimate prevents it from converging—SGD is not a variance-reduced method[17]—although larger w increases precision since it causes a larger fraction of the dataset to be factored in. For SAG, which is variance-reduced, convergence stops as a result of not factoring in samples stored by workers that are straggling over many subsequent iterations (see Section 5.1). For $w = N$, SAG converges to the optimum since all workers participate in each iteration, at the expense of increased latency, i.e., there is a trade-off between straggler-resiliency and convergence.

DSAG extends SAG by incorporating stale results, and, as a result, converges to the optimum even when $w < N$, allowing it to achieve both low latency and high precision in the presence of stragglers. In this instance, DSAG with $w = 10$ is the fastest of all methods considered for both PCA and logistic regression, except for when solving PCA to within a precision of about 10^{-3}, in which case SGD is faster. In particular, DSAG with $w = 10$ achieves a rate of convergence comparable to that of SAG with $w = N$, but reduces latency by an amount that is proportional to the amount of latency variability. For example, for PCA, DSAG with $w = 10$ is between about 20% (for a suboptimality gap of 10^{-4}) and 30% (for a suboptimality gap of 10^{-8} or lower) faster than SAG with $w = N$, and, for logistic regression, DSAG with $w = 10$ is about 30% faster than SAG when the suboptimality gap is 10^{-4} or lower. Finally, for both PCA and logistic regression, the straggler resiliency afforded by coding is canceled out by the higher computational load. Here, we consider a code rate $r = 45/49$, which we find yields lower latency compared to the lower rates typically used in coded computing (e.g., in [4], [5], [6], [21], [22]).

Next, we evaluate the proposed load-balancer, which we apply to DSAG, SAG, and SGD—we refer to the corresponding load-balanced methods as DSAG-LB, SAG-LB, and SGD-LB, respectively. For SAG-LB, to allow for dynamically re-sizing the data partitions, we use the DSAG update rule (see Section 5), except that stale results are discarded, instead of that in [7]. There are two important caveats. First, it takes about 7 and 0.5 seconds for the load-balancer to produce a first solution for PCA and logistic regression, respectively, before which it has no effect (it is slower for PCA due to the larger number of subpartitions). Second, load-balancing can reduce precision when the suboptimality gap is low due to cache invalidation (see Example 1).[18] This problem is especially severe when the number of subpartitions is large relative to the total number of iterations (as is the case for the PCA problem we consider) since a larger fraction of the overall optimization time is spent before the cache is re-populated. As a result, load-balancing does not result in a speedup for PCA. However, for DSAG with $w = 10$ applied to logistic regression, load-balancing results in about 30% to 40% lower latency when the suboptimality gap is between 10^{-6} and 10^{-12}. Interestingly, the primary mechanism by which load-balancing reduces latency is by increasing the average number of workers that respond within the 2% latency tolerance (see Section 5.1), which allows it to reduce the workload for all workers without reducing the expected overall contribution (see Section 6.2). Further, load-balancing improves the precision of SAG with $w < N$ since the probability of each worker participating becomes more uniform.

7.3 Performance on AWS

Here, we consider performance on a cluster composed of $N = 100$ workers on AWS. To ensure that the results are representative, we use a fresh set of virtual machine instances for each set of experiments. While the results on AWS are similar to those on eX3, there are a few important differences. First, communication latency is about an order of magnitude higher on AWS compared to eX3, whereas computation latency is about 10% to 30% higher, depending on the scenario (when accounting for the fact that the per-worker computational load is about half that of eX3). We show the approximate latency range for the stochastic methods without load-balancing in Table 1. As a result, the performance advantage of the stochastic methods compared to GD and coded computing is reduced somewhat, although they are still about twice as fast.

Second, latency is noisier on AWS, with workers experiencing unpredictable high-latency bursts, which may affect both communication and computation latency. Further, the noise makes up a larger fraction of the overall latency for lower average latency. As a result, the straggler problem is more severe for logistic regression than for PCA, for which each iteration is much slower (see Table 1). In particular, for PCA, DSAG

17. A popular variance reduction technique for SGD is to gradually decrease the stepsize, but doing so reduces the rate of convergence.

18. This problem could be alleviated by disabling load-balancing when close to convergence.

with $w = 10$ is only up to about 10% faster than SAG with $w = N$ (for a suboptimality gap below 10^{-6}), whereas for logistic regression DSAG with $w = 5$ is about 30% faster when the suboptimality gap is 10^{-4} or lower.

Finally, the level of static variation in latency between workers is smaller on AWS than on eX3 (which we modeled after Azure). Hence, the advantage of load-balancing is smaller—about 10% to 15% for DSAG-LB with $w = 20$ compared to DSAG with $w = 10$ (which is fastest when not load-balancing), for logistic regression, and up to about 50% faster than SAG with $w = N$.

8 Conclusions

Recently, there has been significant interest in coded computing, which is often motivated by the straggler problem in distributed machine learning and data analytics. However, we find that there are applications for which coded computing reduces performance compared to GD, even when not accounting for the decoding latency, which may be substantial. One issue is that coded computing methods are often designed under the assumption that latency is i.i.d. between workers, which is typically not the case. Further, there are fundamental differences between the distributed computing problem and the communication problem that erasure correcting codes were designed to address. In particular, we find that, for iterative methods, missing information can be substituted by stale information received over previous iterations, with only a marginal reduction to the rate of convergence. In this way, variance-reduced stochastic optimization methods can achieve straggler resiliency without increasing computational complexity, as is the case for coded computing.

In this work, we have proposed DSAG, which alleviates the straggler problem by only waiting for the fastest subset of workers, while integrating the results computed by stragglers in an asynchronous manner. DSAG is based on the SAG method and uses a variance reduction strategy to improve convergence. Further, we have proposed a load-balancing strategy that is able to counter some of the latency variability that exists in distributed computing systems, without moving data between workers. For both PCA and logistic regression, we have shown that DSAG can reduce latency significantly—by up to 50% for logistic regression on AWS, compared to SAG—through a combination of load-balancing and only waiting for the fastest subset of workers.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[2] L. A. Barroso, U. Hölzle, P. Ranganathan, and M. Martonosi, "WSC hardware building blocks," in *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed. San Rafael, CA: Morgan & Claypool Publishers, 2018, ch. 3.

[3] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Trans. Inf. Theory*, vol. 64, no. 3, Mar. 2018.

[4] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proc. Int. Conf. Machine Learning (ICML)*, Sydney, NSW, Australia, Aug. 2017.

[5] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, "Straggler mitigation in distributed optimization through data encoding," in *Proc. Neural Inf. Processing Syst. (NIPS)*, Long Beach, CA, Dec. 2017.

[6] Y. Yang, M. Chaudhari, P. Grover, and S. Kar, "Coding for a single sparse inverse problem," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, Jun. 2018.

[7] M. Schmidt, N. Le Roux, and F. Bach, "Minimizing finite sums with the stochastic average gradient," *Math. Programming*, vol. 162, no. 1, Mar. 2017.

[8] A. Severinson. (2021) DSAG source code and latency traces. [Online]. Available: https://github.com/severinson/DSAG-Paper

[9] A. Severinson, A. Graell i Amat, and E. Rosnes, "Block-diagonal and LT codes for distributed computing with straggling servers," *IEEE Trans. Commun.*, vol. 67, no. 3, Mar. 2019.

[10] A. Severinson, A. Graell i Amat, E. Rosnes, F. Lázaro, and G. Liva, "A droplet approach based on Raptor codes for distributed computing with straggling servers," in *Proc. Int. Symp. Turbo Codes Iterative Inf. Processing (ISTC)*, Hong Kong, China, Dec. 2018.

[11] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," in *Proc. Neural Inf. Processing Syst. (NIPS)*, Long Beach, CA, Dec. 2017.

[12] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Vail, CO, Jun. 2018.

[13] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD," in *Proc. Int. Conf. Artificial Intelligence Stat. (AISTATS)*, Lanzarote, Canary Islands, Spain, Apr. 2018.

[14] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Aachen, Germany, Jun. 2017.

[15] M. Fahim and V. R. Cadambe, "Numerically stable polynomially coded computing," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Paris, France, Jul. 2019.

[16] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *IEEE Trans. Inf. Theory*, vol. 66, no. 1, Jan. 2020.

[17] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, "OverSketch: Approximate matrix multiplication for the cloud," in *Proc. IEEE Int. Conf. Big Data (BigData)*, Seattle, WA, Dec. 2018.

[18] T. Jahani-Nezhad and M. A. Maddah-Ali, "CodedSketch: Coded distributed computation of approximated matrix multiplication," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Paris, France, Jul. 2019.

[19] C.-S. Yang, R. Pedarsani, and A. S. Avestimehr, "Timely-throughput optimal coded computing over cloud networks," in *Proc. ACM Int. Symp. Mobile Ad Hoc Netw. Comput. (MobiHoc)*, Catania, Italy, Jul. 2019.

[20] M. Ye and E. Abbe, "Communication-computation efficient gradient coding," in *Proc. Int. Conf. Machine Learning (ICML)*, Stockholm, Sweden, Jul. 2018.

[21] R. Bitar, M. Wootters, and S. El Rouayheb, "Stochastic gradient coding for straggler mitigation in distributed learning," *IEEE J. Sel. Areas Inf. Theory*, vol. 1, no. 1, May 2020.

[22] H. Wang, Z. Charles, and D. Papailiopoulos, "ErasureHead: Distributed gradient descent without delays using approximate gradient coding," Jan. 2019, arXiv:1901.09671.

[23] P. Richtárik and M. Takáč, "Stochastic reformulations of linear systems: Algorithms and convergence theory," *SIAM J. Matrix Anal. Appl.*, vol. 41, no. 2, Jan. 2020.

[24] F. Hanzely, K. Mishchenko, and P. Richtárik, "SEGA: Variance reduction via gradient sketching," in *Proc. Neural Inf. Processing Syst. (NeurIPS)*, Montréal, QC, Canada, Dec. 2018.

[25] R. M. Gower and P. Richtárik, "Randomized iterative methods for linear systems," *SIAM J. Matrix Anal. Appl.*, vol. 36, no. 4, Jan. 2015.

[26] A. Defazio, F. Bach, and S. Lacoste-Julien, "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives," in *Proc. Neural Inf. Processing Syst. (NIPS)*, Montréal, QC, Canada, Dec. 2014.

[27] C. Calauzènes and N. Le Roux, "Distributed SAGA: Maintaining linear convergence rate with limited communication," May 2017, arXiv:1903.03934.

[28] L. M. Nguyen, J. Liu, K. Scheinberg, and M. Takáč, "SARAH: A novel method for machine learning problems using stochastic recursive gradient," in *Proc. Int. Conf. Machine Learning (ICML)*, Sydney, NSW, Australia, Aug. 2017.

[29] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proc. Neural Inf. Processing Syst. (NIPS)*, Lake Tahoe, NV, Dec. 2013.

[30] E. Gorbunov, K. Burlachenko, Z. Li, and P. Richtárik, "MARINA: Faster non-convex distributed learning with compression," in *Proc. Int. Conf. Machine Learning (ICML)*, Jul. 2021.

[31] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Proc. Neural Inf. Processing Syst. (NIPS)*, Montréal, QC, Canada, Dec. 2015.

[32] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Barcelona, Spain, Dec. 2016.

[33] B. Recht, C. Re, S. Wright, and F. Niu, "HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. Neural Inf. Processing Syst. (NIPS)*, Grenada, Spain, Dec. 2011.

[34] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, and C. Ré, "CYCLADES: Conflict-free asynchronous machine learning," in *Proc. Neural Inf. Processing Syst. (NIPS)*, Barcelona, Spain, Dec. 2016.

[35] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. IEEE Int. Parallel Distributed Processing Symp. (IPDPS)*, Long Beach, CA, Mar. 2007.

[36] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proc. Int. Conf. High Performance Comput., Netw., Storage Analysis (SC)*, Salt Lake City, UT, Nov. 2016.

[37] S. A. Javadi and A. Gandhi, "DIAL: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Columbus, OH, Jul. 2017.

[38] A. K. Maji, S. Mitra, and S. Bagchi, "ICE: An integrated configuration engine for interference mitigation in cloud services," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Grenoble, France, Jul. 2015.

[39] A. Auton *et al.*, "A global reference for human genetic variation," *Nature*, vol. 526, no. 7571, Oct. 2015.

[40] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning," *Nature Commun.*, vol. 5, Jul. 2014.

[41] J. Langguth, X. Cai, and M. Sourouri, "Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures," in *Proc. IEEE Int. Conf. Parallel Distributed Syst. (ICPADS)*, Singapore, Dec. 2018.

[42] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, Bordeaux, France, Apr. 2015.

[43] M. Luksa, *Kubernetes in Action*. Manning, 2017.

[44] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," *Math. Syst. Theory*, vol. 10, no. 1, Dec. 1976.

Paper IV

Coded distributed tracking

Albin Severinson, Eirik Rosnes, and Alexandre Graell i Amat

In Proc. IEEE Global Communications Conference.

The layout has been revised.

Coded Distributed Tracking

Albin Severinson*†, Eirik Rosnes*, and Alexandre Graell i Amat‡*

*Simula UiB, Bergen, Norway
†Department of Informatics, University of Bergen, Bergen, Norway
‡Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Sweden

Abstract

We consider the problem of tracking the state of a process that evolves over time in a distributed setting, with multiple observers each observing parts of the state, which is a fundamental information processing problem with a wide range of applications. We propose a cloud-assisted scheme where the tracking is performed over the cloud. In particular, to provide timely and accurate updates, and alleviate the straggler problem of cloud computing, we propose a coded distributed computing approach where coded observations are distributed over multiple workers. The proposed scheme is based on a coded version of the Kalman filter that operates on data encoded with an erasure correcting code, such that the state can be estimated from partial updates computed by a subset of the workers. We apply the proposed scheme to the problem of tracking multiple vehicles. We show that replication achieves significantly higher accuracy than the corresponding uncoded scheme. The use of maximum distance separable (MDS) codes further improves accuracy for larger update intervals. In both cases, the proposed scheme approaches the accuracy of an ideal centralized scheme when the update interval is large enough. Finally, we observe a trade-off between *age-of-information* and estimation accuracy for MDS codes.

1 Introduction

Tracking the state of a process that evolves over time in a distributed fashion is one of the most fundamental distributed information processing problems, with applications in, e.g., signal processing, control theory, robotics, and intelligent transportation systems (ITS) [1], [2], [3]. These applications typically require collecting data from multiple sources that is analyzed and acted upon in real-time, e.g., to track vehicles in ITS, and rely on timely status updates to operate effectively. The analysis and design of schemes for providing timely updates has received a significant interest in recent years. In a growing number of works, timeliness is measured by the *age-of-information* (AoI) [4], defined as the difference between the current time, t, and the largest generation time of a received message, $U(t)$, i.e., the AoI is $\Delta t = t - U(t)$.

Distributed tracking often entails highly demanding computational tasks. For example, in many previous works the computational complexity of the tasks performed by each node scales with the cube of the state dimension, see, e.g., [2], [5] and references therein. Thus, the proposed schemes are only suitable for low-dimensional processes. A notable exception is the algorithm proposed in [6], where the overall process is split into multiple overlapping subsystems to reduce the computational complexity. However, the algorithm in [6] is based on iterative message passing and potentially requires many iterations to reach consensus, which makes it difficult to provide timely updates.

Offloading computations over the cloud is an appealing solution to aggregate data and speed up demanding computations such that a stringent deadline is met. In [7], a cloud-assisted approach for autonomous driving was shown to significantly improve the response time compared to traditional systems, where vehicles are not connected to the cloud. However, servers in modern cloud computing systems rarely have fixed roles. Instead, incoming tasks are dynamically assigned to servers [8], which offers a high level of flexibility but also introduces significant challenges. For example, so-called *straggling servers*, i.e., servers that experience transient delays, may introduce significant delays [9]. Thus, for applications requiring very timely updates, offloading over the cloud must be done with care.

Recently, the use of erasure correcting codes has been proposed to alleviate the straggler problem in distributed computing systems [10], [11], [12]. In these works, redundancy is added to the computation such that the final output of the computation can later be decoded from a subset of the computed results. Hence, the delay is not limited by the slowest server.

In this paper, we consider a distributed tracking problem where multiple observers each observe parts of the state of the system, and their observations need to be aggregated to estimate the overall state [2], [6]. The goal is to provide timely and accurate information about the state of a stochastic process. An example is tracking vehicles to generate collision warning messages. We propose a cloud-assisted scheme where the tracking is performed over the cloud, which collects data from all observers. In particular, to speed up computations, the proposed scheme borrows ideas from coded distributed computing by distributing the observations over multiple workers, each computing one or more partial estimates of the state of the system.

These partial estimates are finally merged at a monitor to produce an estimate of the overall state. To make the system robust against straggling servers, which may significantly impair the accuracy of the estimate unless accounted for, redundancy is introduced via the use of erasure correcting codes. In particular, the observations are encoded before they are distributed over the workers to increase the probability that the information is propagated to the monitor. A salient contribution of the paper is a coded filter based on the Kalman filter [1] that takes coded observations as its input and returns a state estimate encoded with an erasure correcting code. Hence, the monitor can obtain an overall estimate from a subset of the partial estimates via a decoding operation. We apply the proposed scheme to the problem of tracking multiple vehicles using repetition codes and random maximum distance separable (MDS) codes. We show that replication achieves significantly higher accuracy than the corresponding uncoded scheme and that MDS codes further improve accuracy for larger update intervals. Notably, both schemes approach the accuracy of an ideal centralized scheme for large enough update intervals. Finally, for MDS codes we observe a trade-off between AoI and accuracy, with update intervals shorter than some threshold leading to significantly lower accuracy.

2 System Model and Preliminaries

We consider the problem of tracking the state of a stochastic process over time in a distributed setting. The state at time step t is represented by a real-valued vector $\boldsymbol{x}_t$ of length d and evolves over time according to

$$\boldsymbol{x}_t = \boldsymbol{F}\boldsymbol{x}_{t-1} + \boldsymbol{q}_t,$$

where $\boldsymbol{F}$ is the matrix representing the state transition model and $\boldsymbol{q}_t$ is a noise vector drawn from a zero-mean Gaussian distribution with covariance matrix $\boldsymbol{Q}$. We denote by $\hat{\boldsymbol{x}}_t$ the state estimate at time t and we measure the accuracy of the estimate by its root mean squared error (RMSE).

At each time step, a set of N_o observers, $\mathcal{O} = \{o_1, \dots, o_{N_\mathsf{o}}\}$, obtain noisy partial observations of the state of the process. Specifically, the observation made by observer o at time t is represented by the vector

$$\boldsymbol{z}_t^{(o)} = \boldsymbol{H}^{(o)}\boldsymbol{x}_t + \boldsymbol{r}_t^{(o)},$$

where $\boldsymbol{H}^{(o)}$ is a matrix of size $h^{(o)} \times d$ representing the observation model of observer o and $\boldsymbol{r}_t^{(o)}$ is a noise vector drawn from a zero-mean Gaussian distribution with covariance matrix $\boldsymbol{R}^{(o)}$. Furthermore, we denote by $\boldsymbol{z}_t$ the overall observation vector formed by concatenating the observations made by all observers, $\boldsymbol{z}_t^{(o_1)}, \dots, \boldsymbol{z}_t^{(o_{N_\mathsf{o}})}$, and by h the length of $\boldsymbol{z}_t$. Similarly, we denote by $\boldsymbol{H}$ and $\boldsymbol{r}_t$ the overall observation model and noise vector, respectively, such that $\boldsymbol{z}_t = \boldsymbol{H}\boldsymbol{x}_t + \boldsymbol{r}_t$, and by $\boldsymbol{R}$ the covariance matrix of $\boldsymbol{r}_t$. For simplicity we assume that all observations are of equal dimension. We also assume that $h \geq d$ and that the entries of each observation $\boldsymbol{z}_t^{(o)}$ are linear combinations of a small number of entries of $\boldsymbol{x}_t$, i.e., the observation matrices $\boldsymbol{H}^{(o)}$ are sparse, as is the case, e.g., for an observer measuring speed. The observations made by the N_o observers need to be aggregated to estimate the overall state. Since d may be large, the work of aggregating the observations is performed in the cloud over a set of N_w workers, $\mathcal{W} = \{w_1, \dots, w_{N_\mathsf{w}}\}$. We assume that the matrices $\boldsymbol{F}$, $\boldsymbol{Q}$, $\boldsymbol{H}^{(o)}$, and $\boldsymbol{R}^{(o)}$ are known.

2.1 Probabilistic Runtime Model

We assume that workers become unavailable for a random time after completing a computing task, which is captured by the exponential random variable V with probability density function [10], [11]

$$f_V(v) = \begin{cases} \frac{1}{\beta}\mathrm{e}^{-\frac{v}{\beta}} & v \geq 0 \\ 0 & v < 0 \end{cases},$$

where β is used to scale the tail of the distribution, which accounts for transient disturbances that are at the root of the straggler problem. We refer to β as the straggling parameter.

2.2 Distributed Tracking

At time step t, each observer o uploads its observation $\boldsymbol{z}_t^{(o)}$ to the cloud, where the observations are encoded and distributed over the N_w workers. Next, each worker w that becomes available during time step t computes locally one or more partial estimates of the state $\boldsymbol{x}_t$. These partial estimates are forwarded to a monitor, which is responsible for computing the overall estimate of $\boldsymbol{x}_t$, denoted by $\hat{\boldsymbol{x}}_t$, from the partial estimates. Thus, the monitor has access to an updated state estimate at the end of each time step, which can be used for other applications (e.g., to generate collision warning messages in ITS). Finally, the overall estimate is sent back to the workers to be used in the next time step, i.e., we assume that all workers have access to $\hat{\boldsymbol{x}}_{t-1}$ at time step t.

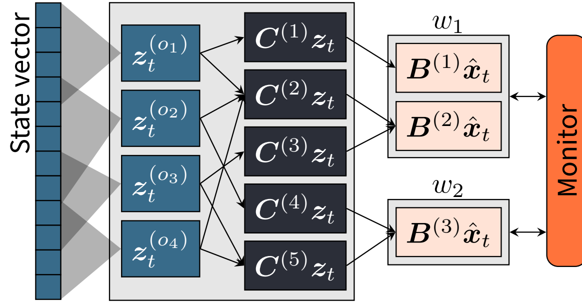


Fig. 1: System overview example. Each of the $N_\mathsf{o} = 4$ observers observes parts of the state, highlighted with a gray cone, and $N_\mathsf{w} = 2$ workers compute 3 coded state estimates from 5 coded observations.

2.3 Kalman Filter

Denote by $\tilde{\boldsymbol{x}}_t$ the prediction of the state at time step t based on the state estimate $\hat{\boldsymbol{x}}_{t-1}$ at time step $t-1$ and the state transition matrix $\boldsymbol{F}$, i.e., $\tilde{\boldsymbol{x}}_t = \boldsymbol{F}\hat{\boldsymbol{x}}_{t-1}$, and by $\tilde{\boldsymbol{P}}_t = \boldsymbol{F}\boldsymbol{P}_{t-1}\boldsymbol{F}^\mathsf{T} + \boldsymbol{Q}$ the covariance matrix of the error $\tilde{\boldsymbol{x}}_t - \boldsymbol{x}_t$, where $(\cdot)^\mathsf{T}$ denotes matrix transposition and $\boldsymbol{P}_{t-1}$ is the covariance matrix of the error $\hat{\boldsymbol{x}}_{t-1} - \boldsymbol{x}_{t-1}$ at time step $t-1$. The Kalman filter is an algorithm for combining the predicted state $\tilde{\boldsymbol{x}}_t$ with an observation $\boldsymbol{z}_t^{(o)} = \boldsymbol{H}^{(o)}\boldsymbol{x}_t + \boldsymbol{r}_t^{(o)}$ to produce an updated state estimate $\hat{\boldsymbol{x}}_t'$ with minimum mean squared error [1]. Let $\tilde{\boldsymbol{y}}_t^{(o)} = \boldsymbol{z}_t^{(o)} - \boldsymbol{H}^{(o)}\tilde{\boldsymbol{x}}_t$ and denote by $\boldsymbol{S}_t^{(o)} = \boldsymbol{R}^{(o)} + \boldsymbol{H}^{(o)}\tilde{\boldsymbol{P}}_t\left(\boldsymbol{H}^{(o)}\right)^\mathsf{T}$ its covariance matrix. Then, the updated state estimate is $\hat{\boldsymbol{x}}_t' = \tilde{\boldsymbol{x}}_t + \boldsymbol{K}_t^{(o)}\tilde{\boldsymbol{y}}_t^{(o)}$, where $\boldsymbol{K}_t^{(o)} = \tilde{\boldsymbol{P}}_t\left(\boldsymbol{H}^{(o)}\right)^\mathsf{T}\left(\boldsymbol{S}_t^{(o)}\right)^{-1}$ is the Kalman gain that determines how the observation should influence the updated estimate. The covariance matrix of the error $\hat{\boldsymbol{x}}_t' - \boldsymbol{x}_t$ is $\boldsymbol{P}_t' = \left(\boldsymbol{I}_d - \boldsymbol{K}_t^{(o)}\boldsymbol{H}^{(o)}\right)\tilde{\boldsymbol{P}}_t$, where $\boldsymbol{I}_d$ is the $d \times d$ identity matrix. If more than one observation is available, the estimate can be improved by setting $\tilde{\boldsymbol{x}}_t \leftarrow \hat{\boldsymbol{x}}_t'$ and $\tilde{\boldsymbol{P}}_t \leftarrow \boldsymbol{P}_t'$ and repeating this procedure. After repeating this procedure for all observations, the final estimate $\hat{\boldsymbol{x}}_t$ is obtained.

3 Proposed Coded Scheme

In this section, we introduce the proposed coded scheme. The key idea is the use of two layers of coding to make the system robust against straggling servers. The first layer consists of encoding the observations and distributing them over multiple workers. More specifically, the overall observation vector $\boldsymbol{z}_t$ is encoded by an (n_C, h) linear erasure correcting code over the reals resulting in the vector $\boldsymbol{C}\boldsymbol{z}_t$, where $\boldsymbol{C}$ is a generator matrix of the code. Next, the elements of $\boldsymbol{C}\boldsymbol{z}_t$ are divided into N_C disjoint subvectors $\boldsymbol{C}^{(1)}\boldsymbol{z}_t, \dots, \boldsymbol{C}^{(N_\mathsf{C})}\boldsymbol{z}_t$, where $\boldsymbol{C}^{(i)}$, $i = 1, \dots, N_\mathsf{C}$, is the corresponding division of the rows of $\boldsymbol{C}$ into submatrices. We denote by $n_\mathsf{C}^{(i)}$ the number of rows of $\boldsymbol{C}^{(i)}$. For the rest of the paper we refer to $\boldsymbol{C}^{(i)}\boldsymbol{z}_t$ as a coded observation. This coding layer increases the probability that the information from an observation propagates to the monitor in case of delays.

The second layer of coding relates to the partial state estimates computed by each worker. Specifically, we propose a coded version of the Kalman filter that takes as its input the overall estimate at the previous time step $\hat{\boldsymbol{x}}_{t-1}$, provided by the monitor, one or more coded observations $\boldsymbol{C}^{(i)}\boldsymbol{z}_t$ from the current time step, and outputs a partial state estimate $\hat{\boldsymbol{x}}_t^{(j)}$. The proposed filter is such that the partial state estimate is equal to the estimate of the regular Kalman filter multiplied by some matrix $\boldsymbol{B}^{(j)}$. Equivalently, it can be seen as an estimate of the state of a process represented by the vector $\boldsymbol{B}^{(j)}\boldsymbol{x}_t$. Let $\boldsymbol{B}$ be a generator matrix of an (n_B, d) linear erasure correcting code over the reals and let $\boldsymbol{B}^{(j)}$ be a submatrix of $\boldsymbol{B}$ of size $n_\mathsf{B}^{(j)} \times d$ such that $\boldsymbol{B}^{(1)}, \dots, \boldsymbol{B}^{(N_\mathsf{B})}$ correspond to a division of the rows of $\boldsymbol{B}$ into $N_\mathsf{B} \le N_\mathsf{C}$ disjoint submatrices. Hence, the partial estimates are symbols of the codeword $\boldsymbol{B}\hat{\boldsymbol{x}}_t$ and the monitor can recover the overall estimate $\hat{\boldsymbol{x}}_t$ from a subset of the partial estimates, ensuring that the monitor has access to timely and accurate estimates even if multiple workers experience delays.

Finally, we associate each coded state estimate $\boldsymbol{B}^{(j)}\hat{\boldsymbol{x}}_t$ with one worker and each coded observation $\boldsymbol{C}^{(i)}\boldsymbol{z}_t$ with one coded state estimate. We index the coded states associated with worker w and the coded observations associated with $\boldsymbol{B}^{(j)}\hat{\boldsymbol{x}}_t$ with the sets $\mathcal{B}^{(w)}$ and $\mathcal{C}^{(j)}$, respectively. In total, worker w receives the set $\{\boldsymbol{C}^{(i)}\boldsymbol{z}_t : i \in \bigcup_{j\in\mathcal{B}^{(w)}} \mathcal{C}^{(j)}\}$ of observations. The overall process is depicted in Fig. 1.

3.1 Coded Update

When a worker w becomes available it first computes the set $\{\boldsymbol{B}^{(j)}\hat{\boldsymbol{x}}_t : j \in \mathcal{B}^{(w)}\}$ of coded state estimates associated with it. The worker also computes a randomly selected subset of the Kalman gains associated with the uncoded filter, which will later be used by the monitor to approximate the covariance matrix $\boldsymbol{P}_t$. Each

coded state estimate $\boldsymbol{B}^{(j)}\hat{\boldsymbol{x}}_t$ is computed from the previous state estimate $\hat{\boldsymbol{x}}_{t-1}$ and the set $\{\boldsymbol{C}^{(i)}\boldsymbol{z}_t : i \in \mathcal{C}^{(j)}\}$ of coded observations associated with it, and is computed independently from the other coded state estimates using the following procedure. First, the worker computes

$$\tilde{\boldsymbol{x}}_t^{(j)} = \left(\boldsymbol{B}^{(j)}\boldsymbol{F}\right)\hat{\boldsymbol{x}}_{t-1}$$

and the covariance matrix

$$\tilde{\boldsymbol{P}}_t^{(j)} = \left(\boldsymbol{B}^{(j)}\boldsymbol{F}\right)\boldsymbol{P}_{t-1}\left(\boldsymbol{B}^{(j)}\boldsymbol{F}\right)^\mathsf{T} + \boldsymbol{B}^{(j)}\boldsymbol{Q}\left(\boldsymbol{B}^{(j)}\right)^\mathsf{T}$$

of the error $\tilde{\boldsymbol{x}}_t^{(j)} - \boldsymbol{B}^{(j)}\boldsymbol{x}_t$. Next, $\tilde{\boldsymbol{x}}_t^{(j)}$ and $\tilde{\boldsymbol{P}}_t^{(j)}$ are combined with the associated observations, i.e., the observations in $\{\boldsymbol{C}^{(i)}\boldsymbol{z}_t : i \in \mathcal{C}^{(j)}\}$, one at a time, to produce $\hat{\boldsymbol{x}}_t^{(j)}$ and the covariance matrix $\boldsymbol{P}_t^{(j)}$ of the error $\hat{\boldsymbol{x}}_t^{(j)} - \boldsymbol{B}^{(j)}\boldsymbol{x}_t$ in the following way. First, consider a matrix $\boldsymbol{A}^{(i,j)}$ such that $\boldsymbol{A}^{(i,j)}\boldsymbol{B}^{(j)} = \boldsymbol{C}^{(i)}\boldsymbol{H}$. Then,

$$\begin{aligned}\boldsymbol{C}^{(i)}\boldsymbol{z}_t &= \boldsymbol{C}^{(i)}\left(\boldsymbol{H}\boldsymbol{x}_t + \boldsymbol{r}_t\right)\\ &= \boldsymbol{C}^{(i)}\boldsymbol{H}\boldsymbol{x}_t + \boldsymbol{C}^{(i)}\boldsymbol{r}_t\\ &= \boldsymbol{A}^{(i,j)}\boldsymbol{B}^{(j)}\boldsymbol{x}_t + \boldsymbol{C}^{(i)}\boldsymbol{r}_t,\end{aligned}$$

i.e., the vector $\boldsymbol{C}^{(i)}\boldsymbol{z}_t$ can be considered as an observation of the state $\boldsymbol{B}^{(j)}\boldsymbol{x}_t$ with observation matrix $\boldsymbol{A}^{(i,j)}$ and observation noise covariance matrix $\boldsymbol{C}^{(i)}\boldsymbol{R}\left(\boldsymbol{C}^{(i)}\right)^\mathsf{T}$. Hence, using an observation $\boldsymbol{C}^{(i)}\boldsymbol{z}_t$, a partial coded state estimate $\hat{\boldsymbol{x}}_t^{\prime(j)}$ and the covariance matrix $\boldsymbol{P}_t^{\prime(j)}$ of the error $\hat{\boldsymbol{x}}_t^{\prime(j)} - \boldsymbol{B}^{(j)}\boldsymbol{x}_t$ can be obtained as

$$\hat{\boldsymbol{x}}_t^{\prime(j)} = \tilde{\boldsymbol{x}}_t^{(j)} + \boldsymbol{K}_t^{(i,j)}\tilde{\boldsymbol{y}}_t^{(i,j)}, \tag{1}$$

$$\boldsymbol{P}_t^{\prime(j)} = \left(\boldsymbol{I}_{n_\mathsf{B}^{(j)}} - \boldsymbol{K}_t^{(i,j)}\boldsymbol{A}^{(i,j)}\right)\tilde{\boldsymbol{P}}_t^{(j)}, \tag{2}$$

where

$$\begin{aligned}\tilde{\boldsymbol{y}}_t^{(i,j)} &= \boldsymbol{C}^{(i)}\boldsymbol{z}_t - \boldsymbol{A}^{(i,j)}\tilde{\boldsymbol{x}}_t^{(j)},\\ \boldsymbol{K}_t^{(i,j)} &= \tilde{\boldsymbol{P}}_t^{(j)}\left(\boldsymbol{A}^{(i,j)}\right)^\mathsf{T}\left(\boldsymbol{S}_t^{(i,j)}\right)^{-1},\\ \boldsymbol{S}_t^{(i,j)} &= \boldsymbol{C}^{(i)}\boldsymbol{R}\left(\boldsymbol{C}^{(i)}\right)^\mathsf{T} + \boldsymbol{A}^{(i,j)}\tilde{\boldsymbol{P}}_t^{(j)}\left(\boldsymbol{A}^{(i,j)}\right)^\mathsf{T}.\end{aligned}$$

Next, we let $\tilde{\boldsymbol{x}}_t^{(j)} \leftarrow \hat{\boldsymbol{x}}_t^{\prime(j)}$ and $\tilde{\boldsymbol{P}}_t^{(j)} \leftarrow \boldsymbol{P}_t^{\prime(j)}$ and repeat (1) and (2) for another observation until all observations in $\{\boldsymbol{C}^{(i)}\boldsymbol{z}_t : i \in \mathcal{C}^{(j)}\}$ have been used, at which point the coded state estimate $\hat{\boldsymbol{x}}_t^{(j)}$ has been computed. The covariance matrix $\boldsymbol{P}_t^{(j)}$ is only needed for computing $\hat{\boldsymbol{x}}_t^{(j)}$ and is discarded at this point. The worker repeats the above procedure for each coded state estimate $\hat{\boldsymbol{x}}_t^{(j)}$, $j \in \mathcal{B}^{(w)}$, assigned to it. Once finished, the worker separately computes the Kalman gain of the uncoded filter $\boldsymbol{K}_t^{(o)}$, as explained in Section 2.3, associated with some number N_K of observers o selected uniformly at random from $\mathcal{O}$. Finally, the coded state estimates are sent to the monitor together with the $\boldsymbol{K}_t^{(i,j)}$ and $\boldsymbol{S}_t^{(i,j)}$ matrices and the uncoded Kalman gains computed by the worker, where they are used to recover the overall state estimate $\hat{\boldsymbol{x}}_t$ and the error covariance matrix $\boldsymbol{P}_t$.

3.2 Decoding

At the end of each time step t the monitor attempts to recover $\hat{\boldsymbol{x}}_t$ from the partial coded state estimates $\hat{\boldsymbol{x}}_t^{(j)}$ received from the workers. This corresponds to a decoding operation. Denote by $\mathcal{U}_t$ the set of coded state estimates the monitor receives at time step t and by $\boldsymbol{B}_{\hat{\boldsymbol{x}}_t}$ the vertical concatenation of the generator matrices associated with those estimates. To decode, the monitor needs to solve for $\hat{\boldsymbol{x}}_t$ in $\boldsymbol{B}_{\hat{\boldsymbol{x}}_t}\hat{\boldsymbol{x}}_t = \boldsymbol{y}_{\hat{\boldsymbol{x}}_t}$, where $\boldsymbol{y}_{\hat{\boldsymbol{x}}_t}$ is the vertical concatenation of the vectors $\hat{\boldsymbol{x}}_t^{(j)}$ in $\mathcal{U}_t$. However, there are two issues that need to be addressed before solving for $\hat{\boldsymbol{x}}_t$. First, due to the dependence structure of the tracking problem and the coding introduced, the elements of $\boldsymbol{y}_{\hat{\boldsymbol{x}}_t}$ will in general be correlated and have varying variance, which must be accounted for to recover $\hat{\boldsymbol{x}}_t$ optimally. Second, since the local estimates by the workers are noisy, $\boldsymbol{B}_{\hat{\boldsymbol{x}}_t}\hat{\boldsymbol{x}}_t = \boldsymbol{y}_{\hat{\boldsymbol{x}}_t}$ typically does not have an exact solution. We address the first issue by applying a so-called *whitening transform* to the original problem, i.e., we solve for $\hat{\boldsymbol{x}}_t$ in $\boldsymbol{M}_{\hat{\boldsymbol{x}}_t}\boldsymbol{B}_{\hat{\boldsymbol{x}}_t}\hat{\boldsymbol{x}}_t = \boldsymbol{M}_{\hat{\boldsymbol{x}}_t}\boldsymbol{y}_{\hat{\boldsymbol{x}}_t}$, where $\boldsymbol{M}_{\hat{\boldsymbol{x}}_t}$ is a linear transform that has the effect of uncorrelating and normalizing the variance of the elements of $\boldsymbol{y}_{\hat{\boldsymbol{x}}_t}$. The whitening transform $\boldsymbol{M}_{\hat{\boldsymbol{x}}_t}$ is computed from the singular value decomposition of the covariance matrix of $\boldsymbol{y}_{\hat{\boldsymbol{x}}_t}$, which we denote by $\boldsymbol{P}_{\hat{\boldsymbol{x}}_t}$, as in [13]. The covariance matrix $\boldsymbol{P}_{\hat{\boldsymbol{x}}_t}$ is given by [14, Eq. (6.47)], where the covariance matrix, Kalman gain, observation model, and observation noise covariance matrix of the uncoded filter update procedure are replaced by their coded equivalents from Section 3.1. We address the second issue by finding the vector $\hat{\boldsymbol{x}}_t$ that minimizes the ℓ_2-norm of the error, i.e., by solving $\arg\min_{\hat{\boldsymbol{x}}_t} ||\boldsymbol{M}_{\hat{\boldsymbol{x}}_t}\boldsymbol{B}_{\hat{\boldsymbol{x}}_t}\hat{\boldsymbol{x}}_t - \boldsymbol{M}_{\hat{\boldsymbol{x}}_t}\boldsymbol{y}_{\hat{\boldsymbol{x}}_t}||_2$. We achieve this by decoding $\hat{\boldsymbol{x}}_t$ using the LSMR algorithm [15]. The LSMR algorithm is a numerical procedure for solving problems of this type that takes an initial guess of the solution as its input and iteratively improves on the solution until it has converged to within some threshold. We give $\tilde{\boldsymbol{x}}_t$, which the monitor computes from $\hat{\boldsymbol{x}}_{t-1}$

as explained in Section 2.3, as the initial guess since the Euclidean distance between $\hat{\boldsymbol{x}}_t$ and $\tilde{\boldsymbol{x}}_t$ typically is small.

Next, the monitor approximates the error covariance matrix $\boldsymbol{P}_t$ using the following heuristic. First, the monitor computes $\tilde{\boldsymbol{P}}_t$ from $\boldsymbol{P}_{t-1}$ as explained in Section 2.3. Denote by r the maximum rank of $\boldsymbol{P}_{\hat{\boldsymbol{x}}_t}$, i.e., the rank of $\boldsymbol{P}_{\hat{\boldsymbol{x}}_t}$ when all workers are available, and by $r_{\hat{\boldsymbol{x}}_t}$ the rank of the given $\boldsymbol{P}_{\hat{\boldsymbol{x}}_t}$. Now, if $r_{\hat{\boldsymbol{x}}_t} < r$ we assume that the monitor has insufficient information to recover $\hat{\boldsymbol{x}}_t$ optimally and we let $\boldsymbol{P}_t = \tilde{\boldsymbol{P}}_t$. On the other hand, if $r_{\hat{\boldsymbol{x}}_t} = r$, we assume that the monitor has recovered $\hat{\boldsymbol{x}}_t$ optimally, and the monitor computes $\boldsymbol{P}_t$ from $\tilde{\boldsymbol{P}}_t$ using the procedure for a full update of the uncoded filter (see Section 2.3). More formally, denote by $\boldsymbol{K}_{t_o}^{(o)}$ the most recently received Kalman gain corresponding to observer o at time step t_o. Then, the monitor computes $\boldsymbol{P}'_t = \left(\boldsymbol{I}_d - \boldsymbol{K}_{t_o}^{(o)}\boldsymbol{H}^{(o)}\right)\tilde{\boldsymbol{P}}_t$, assigns $\tilde{\boldsymbol{P}}_t \leftarrow \boldsymbol{P}'_t$, and repeats the procedure for each remaining observer $o \in \mathcal{O}$. Finally, we let $\boldsymbol{P}_t \leftarrow \tilde{\boldsymbol{P}}_t$. Note that $\boldsymbol{P}_t$ depends only on the statistical properties of the observations, i.e., it can be computed without access to the observations themselves.

4 Design and Analysis of the Proposed Scheme

We analyze the computational complexity of the proposed coded scheme, design the generator matrices required for the coded filter update, and choose how the computations are distributed over the workers.

4.1 Computational Complexity

We assume that the number of arithmetic operations performed by the workers is dominated by the number of operations needed to invert $\boldsymbol{S}$ when computing the Kalman gain, which requires in the order of n^3 operations, where n is the number of rows and columns of $\boldsymbol{S}$. Since each worker computes N_{K} Kalman gains associated with the uncoded filter in addition to those needed for the coded state estimates, and due to our assumption that all uncoded observations have equal dimension, the overall number of operations performed by the workers can be approximated by $N_{\mathsf{K}}N_{\mathsf{w}}\left(h^{(o)}\right)^3 + \sum_{w\in\mathcal{W}}\sum_{j\in\mathcal{B}^{(w)}}\sum_{i\in\mathcal{C}^{(j)}}\left(n_{\mathsf{C}}^{(i)}\right)^3$.

4.2 Code Design

Here, we propose two strategies for designing the sets $\mathcal{B}^{(w)}$ and $\mathcal{C}^{(j)}$ and the matrices $\boldsymbol{B}^{(1)}, \dots, \boldsymbol{B}^{(N_{\mathsf{B}})}$ and $\boldsymbol{C}^{(1)}, \dots, \boldsymbol{C}^{(N_{\mathsf{C}})}$. The matrices $\boldsymbol{A}^{(i,j)}$ are determined implicitly since $\boldsymbol{A}^{(i,j)}\boldsymbol{B}^{(j)} = \boldsymbol{C}^{(i)}\boldsymbol{H}$. The first design is based on replication, which is a special case of MDS codes, whereas the second is based on random MDS codes.

4.2.1 Replication

This design is based on replicating the tracking task at each worker, i.e., the code rate is $h/n_{\mathsf{C}} = 1/N_{\mathsf{w}}$. More formally, each worker estimates $\boldsymbol{x}_t$, i.e., $N_{\mathsf{B}} = N_{\mathsf{w}}$, $\boldsymbol{B}^{(j)} = \boldsymbol{I}_d$, $j = 1, \dots, N_{\mathsf{B}}$, $\mathcal{B}^{(w_1)} = \{1\}, \dots, \mathcal{B}^{(w_{N_{\mathsf{B}}})} = \{N_{\mathsf{B}}\}$, and each state estimate is computed from the full set of observations, i.e., $N_{\mathsf{C}} = N_{\mathsf{w}}N_{\mathsf{o}}$ and the observation encoding matrices and sets $\mathcal{C}^{(j)}$ associated with each estimate $\hat{\boldsymbol{x}}_t^{(j)} = \hat{\boldsymbol{x}}_t$ are such that $\{\boldsymbol{C}^{(i)}\boldsymbol{z}_t : i \in \mathcal{C}^{(j)}\} = \{\boldsymbol{z}_t^{(o)} : o \in \mathcal{O}\}$. Note that the monitor can recover $\hat{\boldsymbol{x}}_t$ and $\boldsymbol{P}_t$ immediately upon receiving these values from any worker without performing any additional computations. Hence, we let $N_{\mathsf{K}} = 0$ and the overall number of operations performed by the workers is approximately $\frac{N_{\mathsf{o}}}{(h/n_{\mathsf{C}})}\left(h^{(o)}\right)^3$.

4.2.2 Random MDS Coding

This design is based on assigning a large number of coded state estimates of dimension one to each worker, i.e., we let $n_{\mathsf{B}}^{(j)} = 1$, $j = 1, \dots, N_{\mathsf{B}}$. Furthermore, to ensure that the code is *well-conditioned*, i.e., the numerical precision lost due to the coding is low, we generate $\boldsymbol{C}$ by drawing each element independently at random from a standard Gaussian distribution [16]. To satisfy the requirement $\boldsymbol{A}^{(i,j)}\boldsymbol{B}^{(j)} = \boldsymbol{C}^{(i)}\boldsymbol{H}$ we let $\boldsymbol{B}^{(j)} = \boldsymbol{C}^{(i)}\boldsymbol{H}$ and $\boldsymbol{A}^{(i,j)} = \boldsymbol{I}_1$. As a result, $n_{\mathsf{C}}^{(i)} = 1$, $i = 1, \dots, N_{\mathsf{C}}$, and we associate each observation one-to-one with a coded state estimate, i.e., $N_{\mathsf{B}} = N_{\mathsf{C}}$ and $\mathcal{C}^{(1)} = \{1\}, \dots, \mathcal{C}^{(N_{\mathsf{B}})} = \{N_{\mathsf{B}}\}$. Next, we split the coded state estimates as evenly as possible over the N_{w} workers, i.e., some workers are assigned $\lfloor N_{\mathsf{B}}/N_{\mathsf{w}} \rfloor$ estimates and some are assigned $\lceil N_{\mathsf{B}}/N_{\mathsf{w}} \rceil$ estimates. Finally, we let $N_{\mathsf{K}} = \left\lceil \frac{N_{\mathsf{o}}/(h/n_{\mathsf{C}})}{N_{\mathsf{w}}} \right\rceil$, which, since $n_{\mathsf{C}}^{(i)} = 1$, $i = 1, \dots, N_{\mathsf{C}}$, means that the overall number of operations performed by the workers is approximately $\frac{N_{\mathsf{o}}}{(h/n_{\mathsf{C}})}\left(h^{(o)}\right)^3$.

5 Numerical Results

To evaluate the performance of the proposed scheme, we consider a distributed vehicle tracking scenario where N_{w} workers cooperate to track the position of N_{v} vehicles $v_1, \dots, v_{N_{\mathsf{v}}}$ based on observations received from the vehicles. We model the state of each vehicle with a length-4 vector composed of its position and

speed in the longitudinal and latitudinal directions, i.e., the overall state dimension is $d = 4N_\mathsf{v}$. As in [17], we assume that the state transition matrix of a single vehicle is

$$\boldsymbol{F}_\mathsf{v} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and that the associated covariance matrix is

$$\boldsymbol{Q}_\mathsf{v} = \boldsymbol{V} \begin{bmatrix} \sigma_\mathsf{a} & 0 \\ 0 & \sigma_\mathsf{a} \end{bmatrix} \boldsymbol{V}^\mathsf{T}, \quad \text{with } \boldsymbol{V} = \begin{bmatrix} \Delta t^2/2 & 0 \\ 0 & \Delta t^2/2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix}.$$

Hence, the combined state transition matrix and covariance matrix for all vehicles is $\boldsymbol{F} = \boldsymbol{I}_{N_\mathsf{v}} \otimes \boldsymbol{F}_\mathsf{v}$ and $\boldsymbol{Q} = \boldsymbol{I}_{N_\mathsf{v}} \otimes \boldsymbol{Q}_\mathsf{v}$, respectively. We assume that each vehicle observes its absolute position, e.g., using global navigation satellite systems (GNSSs), and speed in the longitudinal and latitudinal directions. The corresponding observation matrix is $\boldsymbol{H}_\mathsf{v} = \boldsymbol{I}_4$, with associated covariance matrix $\boldsymbol{R}_\mathsf{GNSS} = \mathrm{diag}(\sigma^2_\mathsf{GNSS}, \sigma^2_\mathsf{GNSS}, \sigma^2_\mathsf{speed}, \sigma^2_\mathsf{speed})$, where $\mathrm{diag}(\cdot)$ denotes the diagonal, or block-diagonal, matrix composed of the arguments of $\mathrm{diag}(\cdot)$ arranged along the diagonal. Furthermore, similar to [17] we assume that each vehicle observes the distance and speed difference in the longitudinal and latitudinal directions relative to a number $s < N_\mathsf{v}$ of other vehicles using, e.g., radar or lidar. By combining these observations in a cooperative manner the accuracy of the vehicle position estimates can be improved compared to a system relying only on GNSS observations. The covariance matrix associated with a relative observation is $\boldsymbol{R}_\mathsf{Rel.} = \mathrm{diag}(\sigma^2_\mathsf{V2V}, \sigma^2_\mathsf{V2V}, \sigma^2_\mathsf{speed}, \sigma^2_\mathsf{speed})$.

For each vehicle v_i, define the matrix $\boldsymbol{U}^{(v_i)}$ of size $(s+1) \times N_\mathsf{v}$, where the first row corresponds to the absolute observation of the vehicle and each of the s remaining rows correspond to an observation relative to another vehicle. The i-th column of $\boldsymbol{U}^{(v_i)}$ has $s+1$ nonzero entries and the remaining columns each have exactly one nonzero entry. For the first row of $\boldsymbol{U}^{(v_i)}$ the i-th entry has value 1, while the remaining entries have value 0. For each of the remaining rows the i-th entry has value -1 and one other entry corresponding to the observed vehicle has value 1. For example, if $s = 2$ and vehicle v_i can observe vehicles v_j and v_k the second and third row will have value 1 in column j and k, respectively. Then, the observation matrix for vehicle v_i is $\boldsymbol{H}^{(v_i)} = \boldsymbol{U}^{(v_i)} \otimes \boldsymbol{H}_\mathsf{v}$. The corresponding observation noise covariance matrix is $\mathrm{diag}\left(\boldsymbol{R}_\mathsf{GNSS}, \boldsymbol{I}_s \otimes \boldsymbol{R}_\mathsf{Rel.}\right)$. Finally, $\boldsymbol{U}^{(v_i)}$ is generated for one vehicle at a time such that the first vehicle observes vehicles $v_2, \ldots, v_{s+1}$, and, in general, vehicle v_i observes vehicles $v_{(j \bmod N_\mathsf{v})+1}$, $j = i, \ldots, i+s-1$.

We compare the performance of the proposed scheme with that of an ideal centralized scheme where the monitor has unlimited processing capacity and processes all observations itself using the procedure in Section 2.3. We also compare against the performance of an uncoded scheme, where each observation is processed by a single worker with no coding. More formally, we divide the N_o observations as evenly as possible over the N_w workers, assigning $\lfloor N_\mathsf{o}/N_\mathsf{w} \rfloor$ observations to some workers and $\lceil N_\mathsf{o}/N_\mathsf{w} \rceil$ observations to the remaining workers. Next, each worker estimates $\boldsymbol{x}_t$ using the uncoded update procedure given in Section 2.3. For this scheme, the monitor estimate is equal to the average of the estimates received from the workers at each time step, i.e., $\hat{\boldsymbol{x}}_t$ is the average of the estimates in $\mathcal{U}_t$ and $\boldsymbol{P}_t$ is the average of the corresponding covariance matrices.

We consider the vehicle tracking problem described above with $\sigma_\mathsf{a} = 0.3$, $\sigma_\mathsf{GNSS} = 2$, $\sigma_\mathsf{V2V} = 0.5$, and $\sigma_\mathsf{speed} = 10$. For all schemes, we run 10 simulations, each of $T = 10000$ time steps, and compute the RMSE of the position estimate at each time step. More specifically, we denote by $\boldsymbol{x}_{\mathsf{p},t}$ and $\hat{\boldsymbol{x}}_{\mathsf{p},t}$ the vectors composed of the entries of $\boldsymbol{x}_t$ and $\hat{\boldsymbol{x}}_t$ corresponding to position, e.g., entries $1, 2, 5, 6$ if $N_\mathsf{v} = 2$, and compute $m_t \triangleq \sqrt{\frac{1}{d/2} \boldsymbol{e}_{\mathsf{p},t} \boldsymbol{e}^\mathsf{T}_{\mathsf{p},t}}$, where $\boldsymbol{e}_{\mathsf{p},t} = \hat{\boldsymbol{x}}_{\mathsf{p},t} - \boldsymbol{x}_{\mathsf{p},t}$, for $t = 1, \ldots, T$. Next, for each simulation, to avoid any initial transients, we discard the first $t_0 - 1$ samples $m_1, \ldots, m_{t_0 - 1}$. We let t_0 be the smallest value such that

$$\frac{|\bar{m}_{t_0:t_\mathsf{m}} - \bar{m}_{(t_\mathsf{m}+1):T}|}{\max\left(\bar{m}_{t_0:t_\mathsf{m}}, \bar{m}_{(t_\mathsf{m}+1):T}\right)} \leq 0.1,$$

where $t_\mathsf{m} = t_0 + \lfloor (T - t_0)/2 \rfloor$ and $\bar{m}_{t_1:t_2}$ denotes the mean of $m_{t_1}, \ldots, m_{t_2}$. Finally, we plot the 90-th percentile of the RMSE of the position estimate over the concatenation of the remaining samples from all simulations.

In Fig. 2 (left), we show the 90-th percentile of the RMSE of the position as a function of the update interval Δt for replication and random MDS codes with rates $1/2$ and $1/3$. For replication the code rate is $h/n_\mathsf{C} = 1/N_\mathsf{w}$ (see Section 4.2), i.e., the number of workers is $N_\mathsf{w} = 2$ and $N_\mathsf{w} = 3$ for rates $1/2$ and $1/3$, respectively. MDS codes support an arbitrary number of workers and we let $N_\mathsf{w} = 16$ for this design. We show the RMSE for $0.01 \leq \Delta t \leq 0.25$ since several applications in ITS require an AoI in this range [3]. There are $N_\mathsf{v} = 10$ vehicles, each observing $s = 5$ other vehicles, and the straggling parameter is $\beta = 0.1$, i.e., workers become unavailable for 0.1 seconds on average after a filter update. Here, replication improves accuracy significantly compared to the uncoded scheme, with a 90-th percentile RMSE of about 0.27 and 0.25 meters for code rates $1/2$ and $1/3$, respectively, when $\Delta t = 0.1$. MDS codes improve the accuracy further at

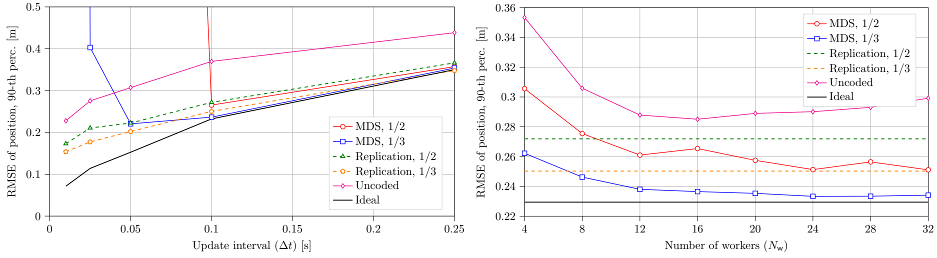


Fig. 2: 90-th percentile of the RMSE of the position over 10 simulations, each of $T = 10000$ samples. On the left, as a function of Δt for $N_\mathsf{v} = 10$, $s = 5$, $N_\mathsf{w} = 16$ (for MDS codes), and $\beta = 0.1$. On the right, as a function of N_w for $N_\mathsf{v} = 10$, $s = 5$, $\Delta t = 0.1$, and $\beta = 0.1$.

this point, with about a 2.4% and 5.5% smaller error when compared at code rates $1/2$ and $1/3$, respectively. Finally, for MDS codes we observe a trade-off between AoI and accuracy, with update intervals shorter than some threshold, e.g., $\Delta t = 0.05$ for code rate $1/3$, leading to a higher RMSE since the probability of the monitor collecting enough coded state estimates to decode $\hat{\boldsymbol{x}}_t$ approaches zero when $\Delta t \rightarrow 0$.

In Fig. 2 (right), we show the 90-th percentile of the RMSE of the position for random MDS codes as a function of the number of workers N_w for $N_\mathsf{v} = 10$ vehicles, each observing $s = 5$ other vehicles, $\Delta t = 0.1$, and $\beta = 0.1$. We also show the error of replication (with N_w fixed to 2 and 3 for code rates $1/2$ and $1/3$, respectively) and the uncoded and ideal schemes. The accuracy of the design based on MDS codes generally improves with N_w, since the variance of the fraction of workers available in each time step decreases. In some cases, e.g., for code rate $1/2$ and $N_\mathsf{w} = 28$, the error increases since the fraction of servers needed to decode $\hat{\boldsymbol{x}}_t$ may increase if the number of coded state estimates does not divide evenly over the workers. Here, the error of MDS codes is lower than that of replication when $N_\mathsf{w} \geq 12$ and $N_\mathsf{w} \geq 8$ for code rates $1/2$ and $1/3$, respectively. We also observe that the performance does not improve significantly beyond some number of workers.

6 Conclusion

We presented a novel scheme for tracking the state of a process in a distributed setting, which we refer to as coded distributed tracking. The proposed scheme extends the idea of coded distributed computing to the tracking problem by considering a coded version of the Kalman filter, where observations are encoded and distributed over multiple workers, each computing partial state estimates encoded with an erasure correcting code, which alleviates the straggler problem since missing results can be compensated for. The proposed coded schemes achieves significantly higher accuracy than the uncoded scheme and approaches the accuracy of an ideal centralized scheme when the update interval is large enough. We believe that coded distributed tracking can be a powerful alternative to previously proposed approaches.

Acknowledgment

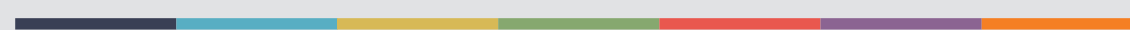
The authors would like to thank Prof. Henk Wymeersch for fruitful discussions and insightful comments.

References

[1] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Trans. ASME J. Basic Eng.*, vol. 82, no. 1, pp. 35–45, Mar. 1960.
[2] R. Olfati-Saber, "Distributed Kalman filtering for sensor networks," in *Proc. IEEE Conf. Decision Control (CDC)*, New Orleans, LA, 2007.
[3] P. Papadimitratos, A. de La Fortelle, K. Evenssen, R. Brignolo, and S. Cosenza, "Vehicular communication systems: Enabling technologies, applications, and future outlook on intelligent transportation," *IEEE Commun. Mag.*, vol. 47, no. 11, pp. 84–95, Nov. 2009.
[4] R. D. Yates and S. Kaul, "Real-time status updating: Multiple sources," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Cambridge, MA, 2012.
[5] M. S. Mahmoud and H. M. Khalid, "Distributed Kalman filtering: a bibliographic review," *IET Control Theory Appl.*, vol. 7, no. 4, pp. 483–501, Mar. 2013.
[6] U. A. Khan and J. M. F. Moura, "Distributing the Kalman filter for large-scale systems," *IEEE Trans. Signal Process.*, vol. 56, no. 10, pp. 4919–4935, Oct. 2008.
[7] S. Kumar, S. Gollakota, and D. Katabi, "A cloud-assisted design for autonomous driving," in *Proc. Workshop Mobile Cloud Comput. (MCC)*, Helsinki, Finland, 2012.
[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. Eur. Conf. Computer Syst. (EuroSys)*, Bordeaux, France, 2015.
[9] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Symp. Oper. Syst. Design Implement. (OSDI)*, San Francisco, CA, 2004.
[10] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *Proc. Workshop Network Coding Appl. (NetCod)*, Washington, DC, 2016.

[11] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” *IEEE Trans. Inf. Theory*, vol. 64, no. 3, pp. 1514–1529, Mar. 2018.
[12] A. Severinson, A. Graell i Amat, and E. Rosnes, “Block-diagonal and LT codes for distributed computing with straggling servers,” *IEEE Trans. Commun.*, vol. 67, no. 3, pp. 1739–1753, Mar. 2019.
[13] A. Kessy, A. Lewin, and K. Strimmer, “Optimal whitening and decorrelation,” *The American Stat.*, vol. 72, no. 4, pp. 309–314, Nov. 2018.
[14] S. Polavarapu, “The Kalman filter,” Dept. of Physics, University of Toronto, Toronto, ON, Canada, Tech. Rep., 2004. [Online]. Available: http://www.atmosp.physics.utoronto.ca/PHY2509/ch6.pdf
[15] D. C.-L. Fong and M. Saunders, “LSMR: An iterative algorithm for sparse least-squares problems,” *SIAM J. Sci. Comput.*, vol. 33, no. 5, pp. 2950–2971, Oct. 2011.
[16] Z. Chen and J. Dongarra, “Numerically stable real number codes based on random matrices,” in *Proc. Int. Conf. Comput. Sci. (ICCS)*, Atlanta, GA, 2005.
[17] G. Soatti, M. Nicoli, N. Garcia, B. Denis, R. Raulefs, and H. Wymeersch, “Implicit cooperative positioning in vehicular networks,” *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 12, pp. 3964–3980, Dec. 2018.

Graphic design: Communication Division, UIB / Print: Skipnes Kommunikasjon AS



uib.no

ISBN: 9788230840726 (print)
9788230868287 (PDF)