Design and Optimization of Heterogeneous Coded Distributed Computing

by

Siyu Zhang

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical and Computer Engineering

Lakehead University

Thunder Bay, Ontario

November 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

Examining Committee Membership

The members listed below served on the Examining Committee for this thesis.

Supervisor:	Dr. Yong Deng
	Department of Software Engineering.
Committee Members:	Dr. M. Mazhar Rathore
	Department of Computer Science.
	Dr. Thangarajah Akilan
	Department of Software Engineering.
Session Chair:	Dr. Yushi Zhou
	Department of Electrical and Computer Engineering.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr.Yong Deng, for his invaluable guidance, encouragement, and continuous support throughout the course of my graduate studies. His deep knowledge, rigorous approach to research, and patient mentorship have been instrumental in shaping the direction of this thesis and in enhancing my academic and professional growth.

I am also thankful to the faculty and staff of the Department of Electrical and Computer Engineering at Lakehead University for providing an excellent academic environment and the resources necessary for my research. I give special thanks to my colleagues and friends for their support and inspiring discussions, which created a collaborative atmosphere and made this journey both productive and enjoyable.

Finally, I would like to express my heartfelt appreciation to my family for their unconditional love, patience, and encouragement. Their understanding and belief in me have been a constant source of strength and motivation throughout the completion of this thesis.

Abstract

The massive increase in data volume in recent years has posed significant challenges for traditional data processing systems. Although distributed computing has been considered as an effective solution, its efficient implementation faces the challenge of the high communication overhead incurred by data exchange (shuffling) between workers. Coded Distributed Computing (CDC) has been proposed by utilizing coded multicasting to reduce the shuffling load. To our best knowledge, existing works on the CDC only consider input files with uniform file size, limiting their practicality in real-world applications. To address this limitation, we propose a Heterogeneous Coded Distributed Computing (HetCDC) scheme to handle input files of nonuniform sizes. We then formulate a joint optimization problem to optimize the file placement and coded shuffling strategies to minimize the shuffling load. Through reformulation, we convert the nonconvex optimization problem into an integer linear programming problem and solve it through the branch-and-cut method. Numerical studies show the proposed HetCDC outperforms existing works. Based on the HetCDC, we further develop a Heterogeneous TeraSort algorithm to improve the sorting time of traditional TeraSort, which is a key building blocks for many big data processing algorithms.

List of Publication

[**1**] S. Zhang, J. Liang and Y. Deng, "Optimization of Decentralized Caching with Improved Coded Delivery Under Heterogeneous File Popularity," in Proc. of the 22nd Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 2025.

[**2**] J. Liang, S. Zhang and Y. Deng, "Optimization of Heterogeneous Coded Distributed Computing with nonuniform Input File Sizes," in Proc. of the 43rd Computing and Communications Conference (IPCCC), Orlando, FL, USA, 2024.Orlando, FL, USA, 2024

Contents

List of Acronym

Acronym	Meaning
$K = \{1, 2, \ldots, K\}$	Set of workers (workers) in the system; $\|K\|$ is the number of workers.
$N = \{1, 2, \ldots, N\}$	Set of input files; $\|N\|$ is the number of files.
$Q = \{1, 2, \ldots, Q\}$	Set of target functions / reduce partitions.
$S \subseteq K,\ S \neq \varnothing$	A nonempty worker subset.
φ_q	Target function $q \in Q$.
$W_k \subseteq Q$	Set of target functions assigned to worker k.
$g_{q,n}$	Map function of φ_q on file n.
$V_{q,n}$	Intermediate value (IV) generated by $g_{q,n}$ from file n.
$M_k \subseteq N$	Files mapped locally by worker k.
F_n	Size of input file n.
m_k	Mapping/storage capacity (quota) of worker k.
$a_{n,S} \in \{0, 1\}$	File–placement indicator: 1 iff file n is placed exclusively on subset S.
A_S	Number of files placed exclusively at subset S.
$L_{k,S\setminus\{k\}}$	Set of IVs required by worker k but generated by workers in $S \setminus \{k\}$.
$l_{k,S\setminus\{k\}}$	Number of IVs worker k needs from $S \setminus \{k\}$.

Acronym	Meaning
$T^i_{k,S\setminus\{k\}}$	IVs assigned to sender $i \in S \setminus \{k\}$ for receiver k.
$t^i_{k,S\setminus\{k\}}$	Number of IVs delivered by worker i to worker k within subset S.
$C_{i,S}$	Coded multicast message transmitted by worker i to $S \setminus \{i\}$ (XOR of sub-messages).
$\lvert C_{i,S}\rvert$	Size of coded message $C_{i,S}$ in IVs (equals $\max_{k\in S\setminus\{i\}} t^i_{k,S\setminus\{k\}}$).
R	Total communication load
$x_{ik,S\setminus\{k\}}$	Auxiliary (epigraph) variable that upper-bounds $t^i_{k,S\setminus\{k\}}$ in the ILP reformulation.
w_q	Number of IVs generated per file (equals $\lvert W_k\rvert$, the number of target functions assigned to a worker).
P_q	Reduce partition handled by worker k.
r	Replication level used in Coded TeraSort examples.
W_n	Content of file n (caching model).
Z_k	Cache content at user/worker k (caching model).
M	Normalized cache size (caching model).
F	File size in the caching model (constant).
d_k	Requested file index of user k in the delivery phase.
$X_{(d_1,\ldots,d_K)}$	Coded multicast message (XOR of requested subfiles) in delivery.
$\oplus$	Bitwise XOR used to form coded messages.
t	Transmitter index inside the holder set S_n for file F_n and partition q; $t \in S_n$ forms the packet $X_t^{(n),q}$.

Acronym	**Meaning**
u	Receiver index in $S_n \setminus \{t\}$ that participates in the XOR formed by transmitter t.
$\tau_{t,u}$	Cursor for receiver u used by transmitter t; selects the next-unsent subpiece $I_u^{(n),q}[\tau_{t,u}]$.
$X_t^{(n),q}$	Coded packet sent by transmitter t for pair (n, q)
$I_k^{(n),q}$	Intermediate value (IV) produced by worker W_k from file F_n that is destined for partition P_q.
B_k	Storage capacity of worker k (in bytes); used by the placement policy under heterogeneous storage.
$\mathcal{L}$	Large-file tier (size class).
$\mathcal{S}$	Small-file tier (size class); groups small files to reduce zero padding.
$\mathcal{M}$	Intermediate-file tier.

List of Figures

List of Tables

Chapter 1

Introduction

The rapid growth of data-intensive applications has fundamentally reshaped modern computing. Traditional single-machine solutions are no longer sufficient to handle the increasing demand, making distributed computing a critical paradigm for handling massive and complex workloads. Despite its advantages, distributed systems also face inherent challenges such as communication bottlenecks, uneven resource utilization, and data heterogeneity, which limit their overall performance. In this chapter, we first discuss the background of our research to provide the necessary context. We then present the motivation for this work, emphasizing the challenges that existing approaches cannot adequately address. Finally, we highlight our main contributions that aim to overcome these limitations.

1.1 Background

With the development of computationally intensive tasks such as Artificial Intelligence(AI), big data analysis, and graph neural networks, the demand for large-scale computation has increased dramatically. These techniques are widely applied in domains such as autonomous driving, recommendation systems, and biomedical analysis, which continuously generate large amounts of data. For example, the DeepSeek V3 large language model was trained in 14.8 trillion tokens [1]. Obviously, the scale of these data-intensive workloads has far exceeded the capabilities of a single machine.

In this context, distributed computing is an effective method for processing large amounts of data. It can connect multiple computing workers via networks to collaboratively execute complex workloads and enhance scalability.

Distributed computing refers to a computational paradigm in which multiple interconnected computing resources collaborate to execute tasks that a single machine cannot efficiently handle. By enabling parallel execution and resource sharing, it provides a scalable and efficient approach to processing large-scale workloads. Over the years, extensive research on parallel processing models, system architectures, and task scheduling strategies has consolidated its theoretical and practical foundations. For example, the authors in [2] proposed the parallel random access machine (PRAM) model, which formalized complexity analysis techniques to characterize interprocessor cooperation and enriched the theoretical framework of the field. The bulk synchronous parallel (BSP) model [3] introduced the notion of a 'superstep', standardized synchronization and communication between compute workers. Then, the authors in [4] proposed the Multi-BSP model by empirically calibrating latency parameters through multicore benchmarking, thereby enhancing real-world applicability. In distributed computing, three prevalent computational models are widely recognized: cluster computing, grid computing, and cloud computing [5].

Cluster Computing: Cluster computing refers to the interconnection of a group of loosely coupled computers through a local area network (LAN), allowing them to cooperate as a unified computing resource. In essence, the system can be viewed as a single logical machine [6]. Cluster systems are widely used in high-performance computing applications, such as scientific simulations, big data analytics, and training machine learning models. Their advantages include low-latency communication, fault tolerance, and ease of scalability in homogeneous environments [7].

Grid Computing: Grid computing aggregates heterogeneous, geographically dispersed resources into a virtual cluster to tackle large-scale problems. It emphasizes cross-domain collaboration via virtual organizations [8], enabling tasks that exceed the capacity of a single supercomputer while remaining suitable for smaller multiuser workloads. Compared with traditional clusters, grids integrate loosely coupled and often untrusted resources and support various hardware combinations [9]; they also allow dynamic resource membership and operate across LAN/MAN/WAN infrastructures with high scalability [10]. We quantify its effectiveness through a resource utilization rate. Large-scale grid systems such as EGEE have demonstrated the practicality of grid computing in real-world e-science collaborations [11].

Cloud Computing: Cloud computing is an internet-based computing paradigm that provides dynamically scalable resources and services on demand, without requiring users to understand the underlying technical details [12]. Users can easily access computing resources such as servers, storage, and applications via the internet. It offers significant advantages, including high scalability, high availability, on-demand services, and low cost, and is widely applied in areas such as enterprise informatization, personal storage, and software development. Compared with other distributed computing models, cluster computing is suited for scenarios requiring high speed and reliability with frequent communication between workers. Grid computing is more appropriate for solving large-scale, cross-domain problems. Cloud computing, with its convenient usage and flexible resource allocation, meets diverse user demands for computing resources. In addition, recent research further explores cloud computing as a foundation for edge computing, serverless infrastructure, and AI as a service [13].

MapReduce

Currently, commonly used distributed computing frameworks are driven by dataflow architectures such as MapReduce [14] and Spark [15]. To simplify distributed computing for developers and support the development of parallel applications by those with limited experience, Google proposed the MapReduce program model in 2004. This model enables developers to perform large-scale data processing through simple operations, without needing to manage intricate details such as parallel execution, load balancing, data partitioning, or fault tolerance. The MapReduce framework was further enhanced with essential capabilities such as automated fault tolerance, speculative task execution, and optimized data locality, enabling its extensive deployment in large-scale data processing scenarios such as web indexing and log analytics [16]. Meanwhile, Spark extended this model by introducing the abstraction of resilient distributed datasets to enhance fault tolerance and iterative computation efficiency [17].

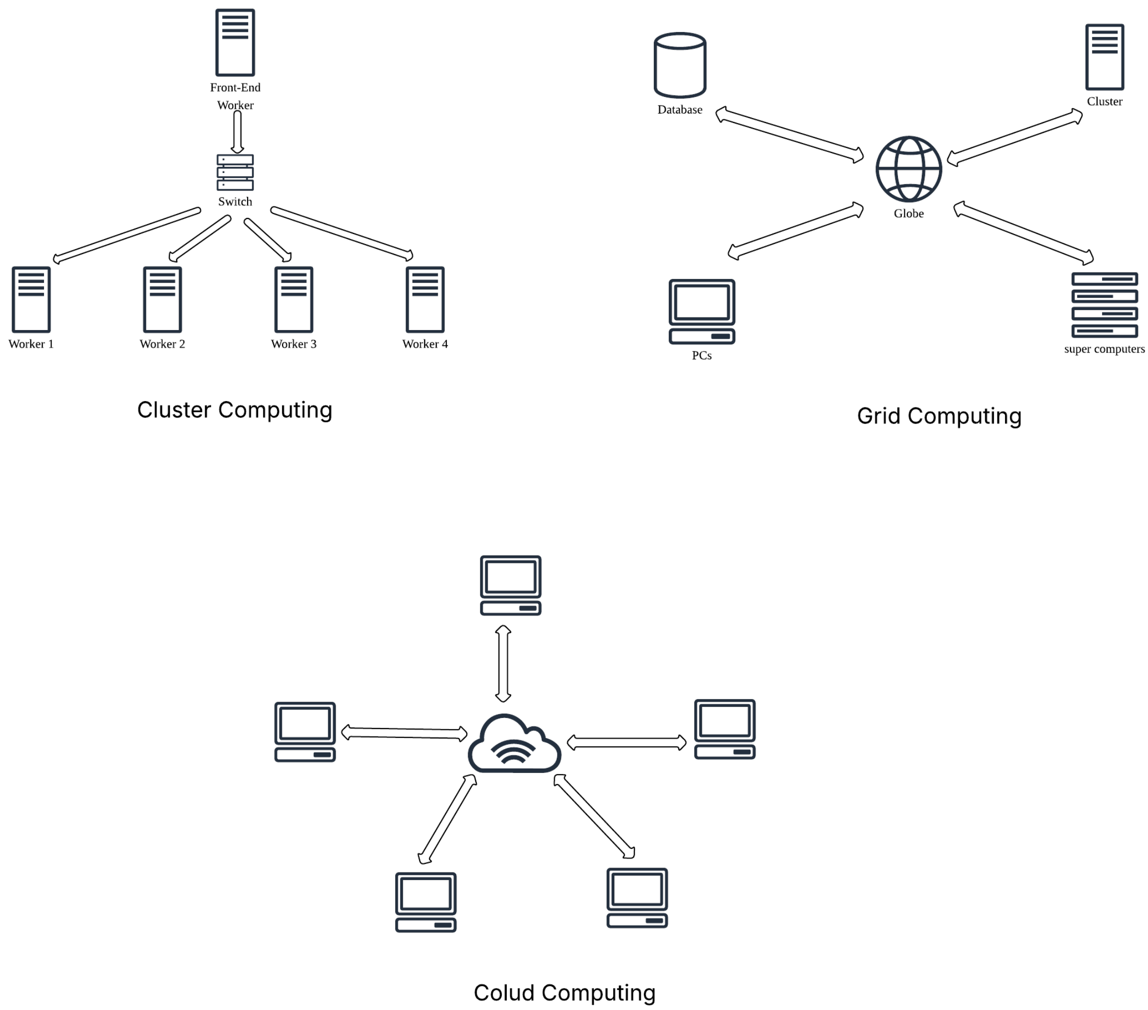


Fig. 1.1: Illustration of Cluster, Grid, and Cloud Computing Paradigms.

When a user invokes MapReduce, a sequence of operations is initiated, and the entire execution workflow proceeds as illustrated in Figure 1.2. In the MapReduce framework, all computations are typically divided into three distinct phases: the Map, Shuffle, and Reduce phases.

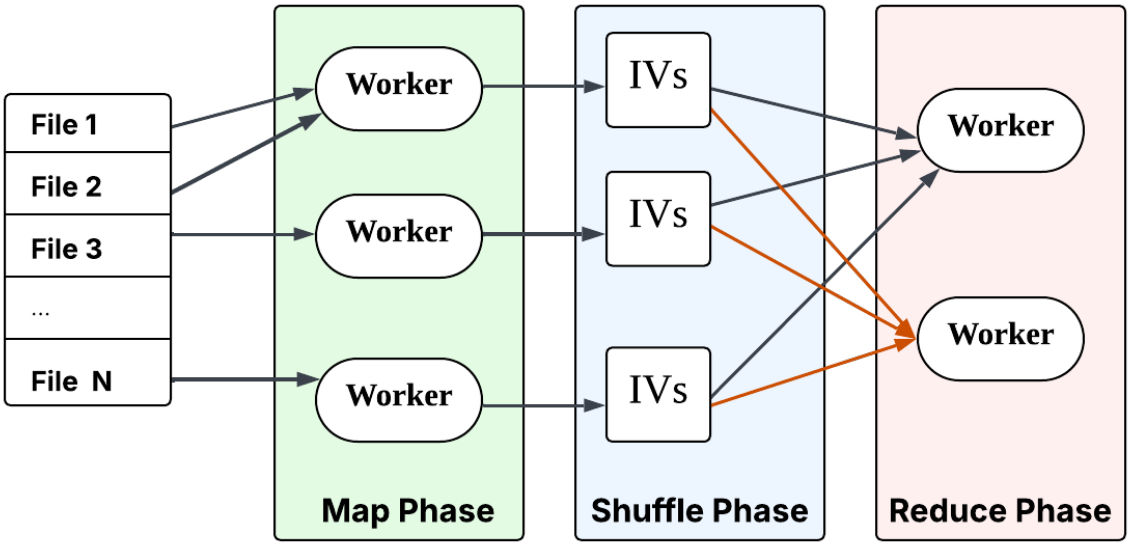


Fig. 1.2: Google MapReduce Working-flow.

Before the Map phase begins, the input data is divided into smaller chunks called splits. Each split represents a portion of the input file and is typically processed by a separate mapper. This division allows for parallel processing of large datasets.

Map: In the Map phase, input files are processed by map functions. Map functions process the input files and partition the results into multiple intermediate files written to the worker's local disk.

Shuffle: The Shuffle phase is between the Map and Reduce phases. It is responsible for redistributing the intermediate values(IVs) such that all values associated with the same key are sent to the same Reduce task. This phase involves transferring data and sorting it by key. The shuffle phase is critical to the correctness and efficiency of the framework.

Reduce: In the Reduce phase, each Reduce task processes all IVs associated with the specific key.

MapReduce has been widely utilized in large-scale data processing tasks. A classical example is the WordCount application, which illustrates the basic map and reduce operations on large text data sets [16]. This example describes how MapReduce divides the input data, processes it in parallel, and aggregates the intermediate results efficiently. Beyond WordCount, MapReduce has been extensively applied in web indexing, log analysis [18], large-scale machine learning [19], and bioinformatics data processing [20], showcasing its scalability and fault tolerance in handling massive datasets.

Now, one of the most widely adopted distributed computing frameworks is Hadoop, developed by the Apache Software Foundation [21]. It consists of two main components: the Hadoop Distributed File System (HDFS), which stores massive data, and the MapReduce programming model, which performs computations over the stored data [14]. In addition to its fundamental architecture, efficient task scheduling and resource allocation are critical to optimizing performance in large-scale systems. For example, the authors in [22] demonstrated that good task assignment strategies can significantly enhance computational throughput in the multiplication of a data-parallel matrix. So data locality aware scheduling, which assigns tasks to workers where the data already resides, has been widely adopted to reduce communication overhead and improve system efficiency, and is now a core principle in frameworks such as Hadoop.

Distributed computing offers parallelism, scalability, and fault tolerance. Parallelism accelerates workloads by executing multiple workers simultaneously; scalability permits capacity expansion; and fault tolerance guarantees continuity even under partial failures. These attributes make distributed computing indispensable for big-data analytics, AI training, scientific computing, and beyond, positioning it as a primary driver of contemporary information technology advancement.

1.2 Motivation

As distributed systems scale, the cost of shuffling communication load across multiple workers becomes a severe bottleneck. For example, during Google's implementation of a 1TB TeraSort task, the shuffle phase took 400 seconds and dominated the overall running time [14]. Empirical evidence demonstrates that the shuffle phase can account for up to 48% of the total execution time in MapReduce workflows. To reduce communication load, researchers have explored techniques that utilize coded multicasting in the shuffle phase.

The concept of coded multicasting originates from the coded caching framework [23], where the placement of content across users' caches is carefully designed so that a single coded transmission can simultaneously serve multiple users with different content requests. This principle of exploiting coding opportunities to deliver data to multiple recipients in one broadcast has inspired its adaptation to distributed computing systems.

Coded Distributed Computing (CDC) extends this idea to the shuffle phase of distributed computing frameworks [24]. By introducing carefully designed computation redundancy during the map phase, CDC creates overlapping IVs between workers, enabling coded multicasting in the shuffle phase. It will allow a single coded message to send the requirements IVs to multiple workers, thereby reducing the total communication load. TeraSort is a large-scale benchmark and sorting application widely used to evaluate the performance of distributed computing systems [25]. It typically operates in a MapReduce framework, where the map phase generates key-value(KV) pairs, the shuffle phase redistributes them based on keys, and the reduce phase outputs the sorted data. Due to the massive data movement involved, the shuffle phase often dominates the execution time, making TeraSort an ideal workload for studying communication-efficient techniques. CDC has been successfully applied to real-world scenarios such as Coded TeraSort [26], achieving up to three-fold reductions in shuffle time.

Although CDC has shown substantial benefits in reducing the shuffle communication load in distributed computing frameworks [24], [26], most existing implementations are designed under the assumption that all input files have a uniform size and that workers possess homogeneous storage, computation, and communication capabilities. These assumptions greatly simplify the design and analysis of CDC schemes, but they rarely hold in practical, large-scale systems.

In real-world scenarios, such as large-scale data analysis and distributed sorting tasks, input files often vary significantly in size due to diverse data sources and preprocessing methods [27]. For example, in Facebook and Microsoft's production MapReduce clusters, file sizes are very uneven: There are many small files, but large files take up most of the total data [28]. Furthermore, computing workers in modern clusters are frequently heterogeneous, comprising a mix of machines with different storage capacities, processing speeds, and network bandwidths [29]. Under such conditions, CDC schemes designed for uniform file sizes and homogeneous workers can lead to load imbalance, suboptimal coding opportunities, and even increased overall execution time.

Addressing these practical limitations is important to fully realize the potential of CDC in modern distributed computing systems. In this work, we propose a HetCDC framework that jointly optimizes file placement and coded shuffling strategies for nonuniform file sizes. By explicitly accounting for file size heterogeneity in optimization, the proposed HetCDC scheme reduces the shuffle communication load in heterogeneous environments.

1.3 Contributions

The main contributions of this thesis are summarized in the following.

- We develop the HetCDC for distributed computing with heterogeneous file sizes, which is commonly seen in practice. In particular, we propose a file placement strategy for the CDC that can efficiently handle an arbitrary number of files of nonuniform sizes.

- We optimize the proposed HetCDC by formulating a joint optimization problem that optimizes file placement and the coded shuffling strategy to minimize the shuffling load. The optimization problem is shown to be a difficult nonconvex problem. Through reformulations, we convert the problem into an equivalent integer linear programming (ILP) problem, which can be solved by the branch-and-cut method. Simulation shows that the optimized HetCDC outperforms the state-of-the-art CDC schemes.

- We further extend the HetCDC design to improve the efficiency of the

TeraSort algorithm, an important Hadoop benchmark for MapReduce computations. In particular, we develop the Heterogeneous Coded TeraSort (HCT) to address the sorting task for files with different sizes. Experiments show that the proposed HCT achieves the lowest sorting time compare with traditional uncoded TeraSort and homogeneous Coded TeraSort algorithms.

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 1 introduces the background of distributed computing and the MapReduce framework, outlining the motivation for addressing heterogeneity in such systems and summarizing the key contributions. Chapter 2 reviews related work, including conventional uncoded methods, coding techniques, coded distributed computing, and the TeraSort benchmark. Chapter 3 presents the proposed HetCDC framework, the system model, and the corresponding optimization formulation. Chapter 4 demonstrates the application of HetCDC to TeraSort and compares its performance with existing approaches. Finally, Chapter 5 concludes the thesis and discusses potential directions for future research.

Chapter 2

Related Works

Distributed computing systems have faced the challenge of improving computational speed. Over the years, two main directions have emerged for speeding up distributed computation. One direction of work focuses on redundancy through task replication. These approaches do not rely on precise scheduling or task partitioning; they will launch redundant executions across multiple workers and proceed with the first completed result. Another direction introduces structured redundancy via coding techniques. By encoding tasks, data, or IVs, these approaches aim to reduce and balance the communication load. In this chapter, we will introduce three major approaches: replication, coding techniques, and coded distributed computing, and conclude with TeraSort as a MapReduce baseline to explore how they speed up distributed computing systems by enhancing efficiency and reducing delays.

2.1 Conventional Uncoded Methods

Replication is an uncoded baseline that runs duplicate copies of the same computation across different workers; the system commits the earliest successful completion or a majority result to improve reliability and reduce runtime variability. Based on early parallel systems [30], replication was adopted in the Shared-Nothing-On-Workstations (SNOW) architecture [31]. They proposed replicating the entire program across multiple workers and taking the output from the first to finish. This approach demonstrated that simple redundancy could lead to measurable improvements in response to performance variability. In distributed computing tasks consisting of many tasks run in parallel, the tasks on the slowest machines (straggling tasks) [32] become the bottleneck in the completion of the task. Although our work is not focused on straggler mitigation, it is worth noting that many replication strategies have been proposed specifically to address the straggler problem in large-scale data-parallel systems.

In 2020, Dolly [29] was introduced as a system that integrates flexible task-level replication into data-parallel frameworks such as Spark[15]. Dolly enforces constraints to avoid excessive overhead. It limits the total number of replicas based on system load and avoids replicating tasks that are short-lived or already near completion. By treating replication as a core part of the scheduling mechanism rather than a reactive response to performance variability, Dolly achieves significant improvements in task completion time, especially in moderately loaded clusters where idle resources are available but underutilized. Although it can reduce running time, Dolly's replication strategy still incurs waste resource overhead, especially when cluster utilization is high. Furthermore, unthinkingly replicating all tasks may lead to inefficiencies if task duration is predictable or the cluster lacks sufficient idle slots. To reduce unnecessary overhead while preserving the latency benefits of replication, a more efficient replication strategy was proposed in cite wang2015straggler. Instead of replicating all tasks at launch time, they proposed a single-fork strategy. It needs to wait until a fixed fraction of tasks (e.g., 80%) have been completed. Then, it replicates only the remaining slowest tasks. This selective replication reduces resource waste on fast tasks while still decreasing running time. In [33], the authors also analyze the replication parameters adjusted in different task duration distributions. Compared to Dolly, this approach is more adaptive and resource-efficient.

Although replication strategies have proven effective in reducing task completion time, they will incur additional resource overhead that may limit their scalability in resource-constrained environments. This limitation motivates the development of more structured and efficient methods to reduce the running time or communication load of distributed computing systems [34].

2.2 Coded Distributed Computing

Coded Distributed Computing (CDC) replaces naive replication with encoding, allowing a single transmission to simultaneously serve multiple workers via coded multicasting or tolerate stragglers by decoding from a subset of results. Two canonical endpoints formalize this idea: the Minimum Bandwidth Code, which repeats Map tasks to minimize shuffle communication [24], and the Minimum Latency Code, which encodes computations to reduce waiting for slow workers [35]. A unified view connects these extremes and anchors the computation–communication balance in the coded-caching perspective [23], [36]. Guided by this trade-off view, the remainder of this subsection surveys CDC along five dimensions: algebraic designs for linear transforms, gradient-based learning, shuffle-phase file placement and coded shuffling, architecture-aware variants, and the application of CDC. It concludes by highlighting nonuniform file sizes as the key open challenge motivating our work.

Algebraic Designs for Linear Transforms: Matrix multiplication is a canonical workload in which CDC can be cleanly formalized. An MDS code encodes data into several blocks such that any prescribed number of returned blocks is sufficient for exact recovery; in practice, this allows the system to tolerate a fixed number of straggling workers with minimal redundancy [37]. In coded computation, input matrices are partitioned, submatrices are MDS encoded, and distributed to workers. The master reconstructs the final product from the first subset of results, which lowers the recovery threshold and straggler-induced latency [35]. Experiments on Amazon EC2 report up to 40% lower average latency and 60% lower tail latency compared to uncoded baselines in m1-small instances [35].

To scale beyond this template, four algebraic families map different points on the threshold–communication–complexity curve: Product codes, Polynomial codes, PolyDot and MatDot, with MatDot achieving a lower recovery threshold by concentrating on relevant cross-products at the expense of higher communication [38]–[41]. On the complexity side, sparsity-aware encodings reduce worker load and decoding time, allowing for nearly linear-time decoding [42]. Short-Dot shortens the dot products of each worker and reports on 32% speed-ups over dense MDS baselines, while exposing a sparsity-recoverability trade-off [43]. The S diagonal codes use a diagonal structure with hybrid peeling and Gaussian elimination to approach near-optimal thresholds with low overhead [44]. The same coding ideas extend beyond matrix–matrix multiplication to coded convolution and the discrete Fourier transform, both common in machine learning and signal processing [45], [46]. Complementing fixed-rate algebraic schemes, rateless designs (LT style) make use of partial work and enable dynamic load balancing in heterogeneous clusters, in contrast to fixed-rate uses of LT, where partial computations may be discarded [47], [48].

Taken together, these algebraic designs motivate the following parts of our review: first, gradient-based learning; then, mechanisms that reduce shuffle communication via file placement and coded shuffling strategy; followed by topology- and architecture-aware variants.

Gradient-based learning: For distributed learning, gradient coding guarantees exact recovery of a batch gradient under erasures or delays. Fractional and cyclic repetition provide baselines [49], while cyclic MDS and Reed–Solomon constructions add structured recovery with balanced masking [50], [51]. Moving toward deployability, the Batch Coupon Collector (BCC) strategy embraces decentralization: it collects any set of sub-batches that cover, reducing wall clock time by using opportunistic returns. Reported results show up to 85.4% faster execution than uncoded methods. 9% over cyclical repetition [52]. Orthogonal to "encode-after-compute," Polynomially Coded Regression (PCR) encodes the data itself so that the master interpolates the final gradient with a lower recovery threshold, thus reducing both the computation and communication time

[53]. These developments jointly indicate that resilience and efficiency can be achieved by coding outputs (gradients) or by moving the coding upstream to the data layer.

File placement and coded shuffling: In MapReduce-like systems, the shuffle phase is often the communication bottleneck, and CDC addresses it along two axes that we will build upon later: file placement and coded shuffling strategy. On the placement side, group-based coded MapReduce creates multicast opportunities while controlling subpacketization [54]; resolvable designs use combinatorial structure to determine subfile counts and placements [55]; systematic storage updates across iterations increase coding opportunities beyond random placement [56]; and locality-aware designs balance data locality with coding gains [57]. The placement–delivery array (PDA) compactly represents both phases in a single array, making construction and analysis more tractable [58]–[60].

On the delivery side, compressed CDC pre-combines IVs before multicast to cut bandwidth [61]; quantized updates (e.g., QSGD) reduce gradient traffic with controlled distortion [62]; pliable index coding and related randomization lower message correlation and redundant transfers across iterations [63]. When tasks are iterative, cross-iteration coding amortizes overhead across steps [64]. For structured pipelines, problem-specific delivery targets the computation graph itself, e.g., distributed graph analytics and multistage MapReduce, to prune redundant sends along the DAG [65]–[67]. A complementary lever is function allocation, which assigns targets to appropriate workers to avoid duplication under mobility or decentralized storage [68]. These techniques jointly reduce the normalized communication load while keeping subpacketization and complexity in check.

Architecture aware variants: CDC abstracts networks as ideal broadcast or co-located workers [35], [36], but practical systems deviate from this model. The topological CDC aligns the code design with realistic fabric in the data center (fat tree) to balance link loads and avoid hot spots

[69]–[71]. To better utilize codebooks in practice, UberShuffle fills in missing entries in encoding tables by reallocating data among workers, with reported reductions in shuffle time of roughly 47. 2% and 35. 7% in different SGD settings compared to a coded shuffling baseline [72]. At the microarchitectural level, multicore-aware CDC exploits intraworker parallelism to fully utilize modern CPUs in homogeneous environments [73]. Together, topology and architecture-aware methods translate the theoretical multicast gains of the CDC into consistent end-to-end improvements.

Application of CDC: CDC ideas now appear in settings where communication is scarce or execution is highly variable. In federated learning, coded/quantized aggregation reduces uplink bandwidth without sacrificing convergence [74], [75]. In heterogeneous edge environments, hierarchical and adaptive strategies accommodate diverse compute and communication budgets between tiers [27], [76], [77]. In serverless platforms, stateless functions and prone execution prone to stragglers motivate CDC-inspired mechanisms that encode intermediate computations to improve resilience and reduce data movement costs [78]–[80]. These applications underscore the versatility of the CDC when both computation and communication resources are constrained or fluctuate over time.

Despite substantial progress, most CDC analyzes model input as equal-size subfiles or intermediate values for analytic tractability (or pad to equal length during XOR), as in Coded MapReduce [24]. In production deployments, however, file and partition sizes are highly nonuniform, e.g., Facebook HDFS traces report that 90% of files are smaller than 15 MB. Hence, the uniform size assumption rarely holds [81]. This mismatch has concrete consequences: subpacketization becomes a practical bottleneck and can create imbalance, motivating a line of low-subpacketization designs [82], [83]; byte-accurate shuffle cost departs from IV-count algebra and requires explicit length alignment or padding for coded multicasting [24]; and the end-to-end completion time is frequently dominated by the largest (skewed) partitions or straggling tasks in MapReduce-style systems [29], [84]. Motivated by this gap, we study CDC under nonuniform file sizes.

Our goal is to design a size-aware file placement and a coded shuffling strategy that preserves coding gains under heterogeneity. We also aim to quantify the computation–communication trade-off under realistic constraints, while reusing and adapting the design levers above (placement, shuffle, encode, and function allocation).

2.3 TeraSort

TeraSort, developed initially as a benchmark for evaluating the performance of MapReduce frameworks, has become a standard for large-scale distributed sorting tasks [25]. It operates in three main stages: sampling via TeraGen, partitioning, and sorting or validation through TeraSort and TeraValidate [85]. In the sampling stage, a subset of keys is extracted to determine partition boundaries; during the shuffle phase, each mapper routes records according to these boundaries, and reducers output globally sorted partitions. Classic results include Yahoo! Hadoop cluster sorting 1 TB in 209 seconds on 910 workers in 2008 [25], improving to 62 seconds on a 1460-worker cluster in 2009 [86], and, in 2013, achieving 1.42 TB in one minute in a 2100-worker cluster [87]. From a theoretical standpoint, TeraSort has been rigorously analyzed as a minimal MapReduce algorithm; appropriate parameterization is shown to guarantee minimality and simplify system-level optimizations [88]. This simplicity and sustained performance have driven TeraSort's widespread academic and industrial use [16], [25], [89].

In practice, the classic total-order partitioner builds a two-level trie on sampled keys and relies on in-memory quicksort, which constrains the sample size; moreover, indexing only the first two bytes of each key can lead to uneven reducer loads under skewed key distributions [85], [90]. To address these issues, follow-up work refines the partitioner: Xtrie replaces the sort step with direct trie insertion, supporting deeper attempts to shrink the per-prefix ranges and improve the load balance [91]. ETrie further reduces memory by grouping ASCII into three classes, effectively reducing the branching factor from 256 to 64; for a two-level trie, this cuts the memory footprint by a factor of 1/16 and enables deeper tries under the same budget, which in turn smooths reducer assignments [91].

Beyond general partitioner refinements, domain-specific pipelines also adapt TeraSort. For climate workloads, an optimized map–sort–reduce flow on historical weather-station data compresses observations into monthly aggregates with near-linear processing cost on a single worker and explores quantum-inspired mechanisms for further speedups [92].

Beyond such domain-specific pipelines, another line of work adapts TeraSort at the network stage via coding. The Coded TeraSort keeps the map–partition–sort operation but introduces structured replication and XOR-based coded multicasting so that one transmission can satisfy multiple reducers, reducing the shuffle load [26]. As this design reduces the dominant shuffle cost; empirically (Table 2.1, 12 GB, $K = 16$, 100 Mbps), raising r further cuts shuffle from 945.72 s to 412.22 s ($r = 3$) and 222.83 s ($r = 5$) while generating total time reductions at 445.56 s (2.16×) and 283.33 s (3.39×). Here, r denotes the replication level. Each input file is placed on exactly r distinct workers.

Table 2.1: Sorting 12 GB Data with $K = 16$ workers and 100 Mbps Network Speed

Scheme	**CodeGen** (sec.)	**Map** (sec.)	**Encode** (sec.)	**Shuffle** (sec.)	**Decode** (sec.)	**Reduce** (sec.)	**Total** (sec.)	**Speedup**
TeraSort	–	1.86	2.35	945.72	0.85	10.47	961.25	–
CodedTeraSort: $r = 3$	6.06	6.03	5.79	412.22	2.41	13.05	445.56	2.16×
CodedTeraSort: $r = 5$	23.47	10.84	8.10	222.83	3.69	14.40	283.33	3.39×

The baseline formulation typically assumes uniform file sizes and balanced replication, which simplifies grouping and analysis, but is limited under heterogeneity. It motivates our setting: we preserve the TeraSort function while introducing heterogeneous input file sizes and capacities through placement-aware groups and length-aligned XOR in the shuffle phase.

Chapter 3

Heterogeneous Coded Distributed Computing with Nonuniform Input File Sizes

This chapter presents the proposed Heterogeneous CDC(HetCDC) framework. We first introduce the system model and review the standard CDC framework as the basis of our work. Then we explain how HetCDC extends CDC to support nonuniform file sizes. This chapter also discusses the design of file placement and coded shuffling strategies, formulates the corresponding optimization problem, and presents the solution method. Finally, simulation results are provided to demonstrate the effectiveness of the framework.

3.1 System Model

We consider the CDC framework with K workers, indexed by $\mathcal{K} = \{1, \ldots, K\}$. The system processes N input files, indexed by $\mathcal{N} = \{1, \ldots, N\}$. In the classical CDC setting, all input files are assumed to be of equal size; The computational task is decomposed into Q target functions $\{\phi_1, \ldots, \phi_Q\}$, indexed by $\mathcal{Q} = \{1, \ldots, Q\}$. Each target function ϕ_q is assigned to a worker; let $\mathcal{W}_k \subseteq \mathcal{Q}$ denote the set of target functions assigned to worker k. As shown in 3.1, the computation follows the Map–Shuffle–Reduce paradigm, as described below.

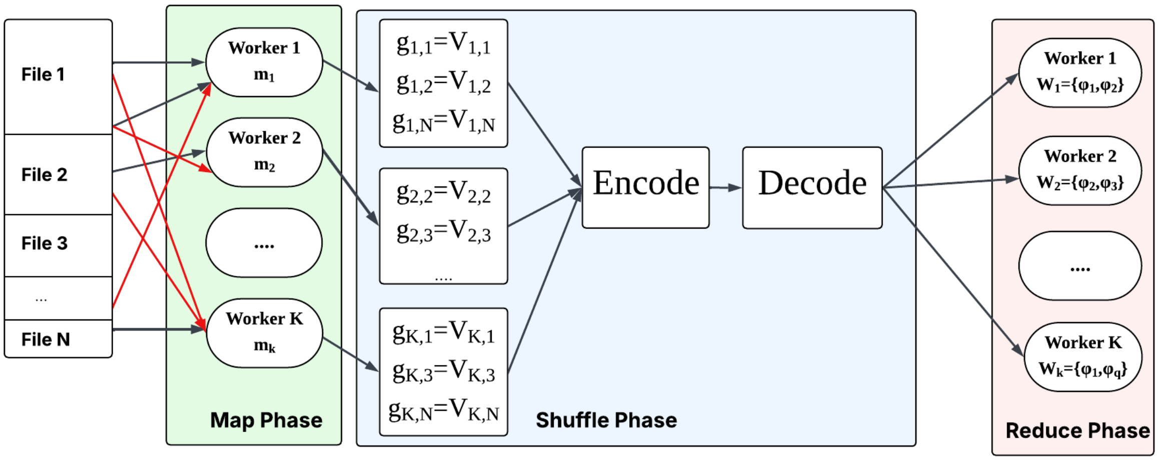


Fig. 3.1: HetCDC System model.

Map: Each target function ϕ_q, $q \in \mathcal{Q}$, is associated with a set of N map functions $\{g_{q,n} : n \in \mathcal{N}\}$. The output of each map function $g_{q,n}$ is an IV, denoted $V_{q,n} = q_{q,n}(W_n)$. Each worker $k \in \mathcal{K}$ is assigned a subset of input files $\mathcal{M}_k \subseteq \mathcal{N}$, and only computes $V_{q,n}$ for $n \in \mathcal{M}_k$, for all q.

Shuffle: To complete the Reduce phase, each worker needs to obtain IVs $V_{q,n}$ for all $n \notin \mathcal{M}_k$ associated with the functions it is assigned. Instead of unicasting each IV individually, CDC exploits coding opportunities: workers transmit coded multicasting messages composed of XOR combinations of IVs, which simultaneously serve the needs of multiple workers. These coded transmissions reduce the total communication load.

Reduce: Once all required intermediate values have been acquired, either through local computation or through decoding received coded transmissions, each worker K computes its assigned functions using the set of IVs

Although the above description corresponds to the standard CDC framework, in later sections, we extend it to handle heterogeneous file sizes and worker capabilities. These generalizations, which form the HetCDC scheme, require redesigned file placement and coded shuffling strategies, which will be detailed in Section 3.3.

3.2 Preliminary

Distributed computing has become a widely adopted model for large-scale data processing. However, communication overhead during the Shuffle phase has emerged as a significant bottleneck that limits the performance of distributed computing frameworks. For example, in the TeraSort benchmark running on the Facebook Hadoop cluster, it was observed that the Shuffle phase alone accounts for approximately 33% of the total execution time [93]. In more intensive applications, such as the training of a convolutional neural network (CNN) [94] that involves millions of updates to model parameters, the amount of computation required becomes significantly larger. To address this problem, CDC was proposed [24], which utilizes coded multicasting during the Shuffle phase to reduce the communication load. The key idea is to introduce redundancy in the Map phase so that IVs can be combined and multicasted efficiently to multiple workers. This coded multicasting strategy enables each transmission to simultaneously satisfy the data requirements of multiple workers, significantly lowering the communication load.

3.2.1 Coded Caching

Coded multicasting was first proposed in coded caching [23]. To better understand how coded multicasting is achieved in this framework, we now briefly introduce the system model and the main procedure of coded caching, which paves the theoretical foundation for CDC. The authors in [95] proposed the system model shown in Figure 3.2. The server stores N files, denoted as $W_1, \ldots, W_N$, each of size F. There are K users in the system, each requesting files from the server. The user k is equipped with a local cache of size MF, denoted by Z_k, which can store a subset of files. Here, $M \in [0, N]$ represents the normalized cache size. All users are connected to the server through a shared, error-free bottleneck link. This shared link carries out both the placement and delivery phases of data transmission.

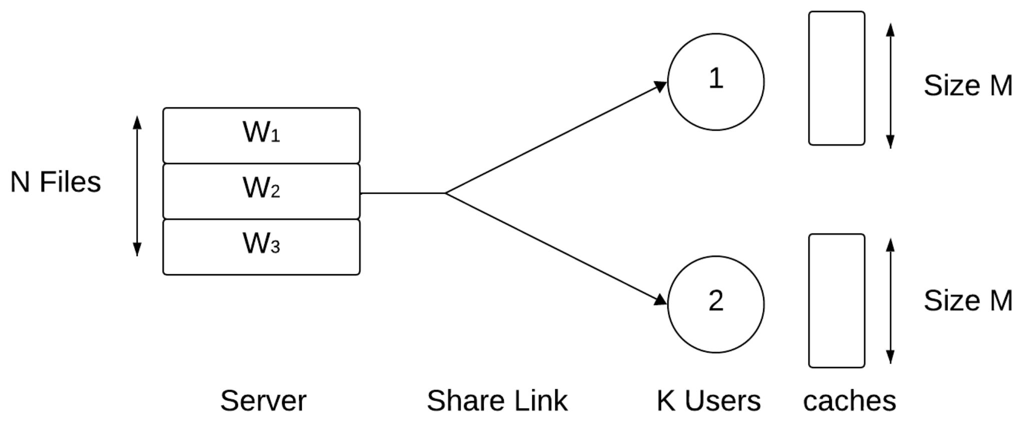


Fig. 3.2: System model for coded caching.

Placement: The placement phase begins by computing the parameter $t = \frac{MK}{N}$, which represents the number of users that will cache a particular subfile. Then, define the set U as the collection of all subsets $T \subseteq [K]$ such that $|T| = t$. For each file W_n, where $n \in [N]$, divide it into subfiles of equal size $\binom{K}{t}$, denoted $W_{n,T}$ for each $T \in U$. These subfiles are indexed by the subsets of users who will cache them. Next, for each user $k \in [K]$, assign to their cache Z_k all subfiles $W_{n,T}$ such that $k \in T$, i.e., user k caches all subfiles whose index set includes k. This completes the placement phase.

Delivery: During the delivery phase, each user k requests a file $d_k \in [N]$. The parameter t is again calculated as $t = \frac{MK}{N}$. Define the set Θ as all subsets $S \subseteq [K]$ such that $|S| = t + 1$. For each subset $S \in \Theta$, the server constructs a multicast transmission $X_{(d_1,\ldots,d_K)}$ by computing the bitwise XOR of the subfiles $W_{d_k,S\setminus\{k\}}$ for each $k \in S$. That is, each transmission is:

$$X_{(d_1,\ldots,d_K)} \leftarrow \bigoplus_{k \in S} W_{d_k,S\setminus\{k\}}$$

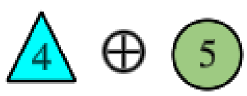
This coded message enables each user $k \in S$ to decode their missing subfile $W_{d_k,S\setminus\{k\}}$ using the coded transmission and their cached subfiles. And this function is called Coded Multicasting.

3.2.2 Coded Distributed Computing

The CDC scheme introduces redundancy in the Map phase by replicating input files across multiple workers to enable coded multicasting, thus achieving a low communication load. To better illustrate Coded Distributed Computing, we now present a simple example3.1 that demonstrates how the Map, Shuffle, and Reduce phases are executed and how coded multicasting helps reduce communication load.

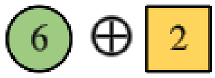
Example 3.1. We consider a distributed computing system with $K = 3$ workers, $N = 6$ input files, and 3 target function(green circle, red squares, blue triangles). The file placement is as follows: Worker 1 stores files $\{1, 2, 3, 4\}$, Worker 2 stores files $\{3, 4, 5, 6\}$, Worker 3 stores files $\{1, 2, 5, 6\}$. Each worker is responsible for computing a function that depends on all input files. The computation proceeds in three phases, as illustrated in figure 3.3.

In the Map phase, each worker computes IVs for all three target functions from its local workers. Green circles are IVs needed by Worker 1. The red squares are the IVs needed by Worker 2. The blue triangles are the IVs needed by Worker 3. For example, Worker 1 in the map phase can generate IV green circles 1, 2, 3, 4, red squares 1, 2, 3, 4, and blue triangles 1, 2, 3, 4. In the Shuffle phase, each worker lacks the IVs corresponding to the files it does not store. For example, in worker 1, it needs to compute the target function 1, indicated by the green circle. So, worker 1 also needs to obtain green circles 5 and 6 IVs from the other 2 workers. So for worker 2, it sent the message to workers 1 and 3.



Because worker 1 needs green circle 5 IV and worker 3 needs blue 4 IV. For worker 1, it already has blue triangle 4. We can use the blue triangle

4 IV to decode the green circle 5 IV. Similarly to worker 3, it can also calculate the blue triangle 4. And worker 3 needs to be sent to worker 1. Finally, worker 1 can get all the IVs it needs..



In the reduce phase, after decoding all missing IVs, each worker can compute its assigned function using IVs from all six files. And in this approach, the communication load is: $3 \times 2 = 6$.

The above example shows how CDC significantly reduces the communication load through coded multicasting. To highlight this advantage more clearly, we now compare it with uncoded distributed computing.

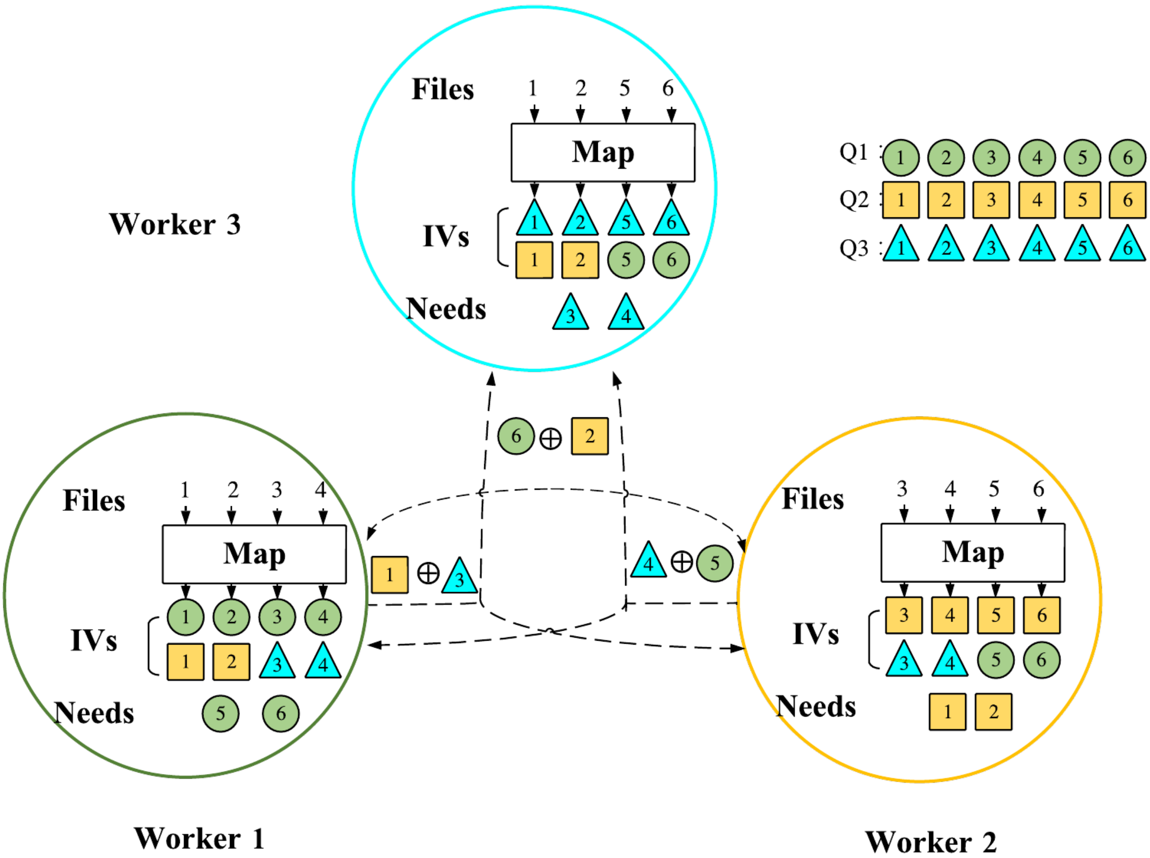


Fig. 3.3: Illustrations of Coded Distributed Computing to compute Q=3 from N=6 using K=3.

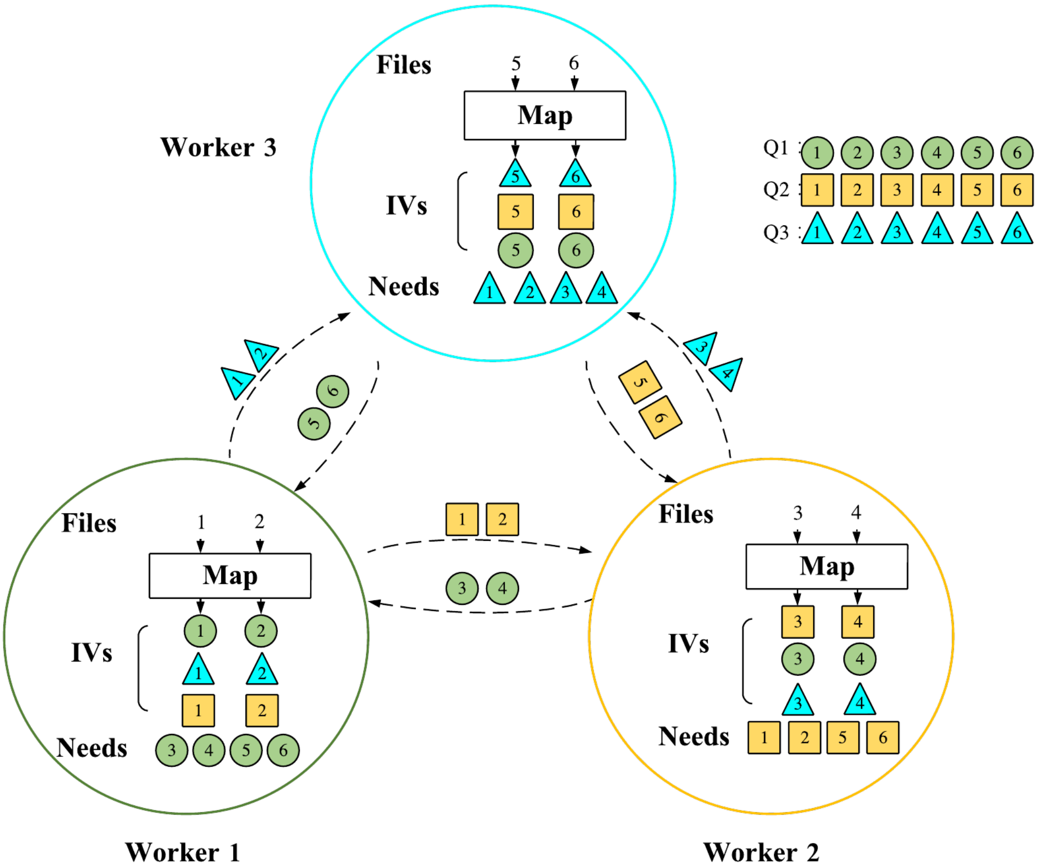


Fig. 3.4: Illustrations of Uncoded Distributed Computing compute Q=3 from N=6 using K=3.

But we consider the same system setup without coded multicasting. In this case, each IV needed by a worker must be transmitted individually from the worker who stores it. For example, in the figure 3.4, worker 1 lacks IVs corresponding to files 3, 4, 5, and 6. So worker 2 sends red square 3 and 4 IVs to worker 1. And worker 3 sends green circle 5 and 6 IVs to worker 1. As a result, worker 1 receives four separate transmissions to complete its assigned target function. Similar transmissions are also required for the other workers. So for an uncoded scheme, the communication load is $3 \times 4 = 12$. Compared to the coded scheme, this results in a significantly higher communication load.

3.3 Heterogeneous Coded Distributed Computing

In the CDC framework, all input files are assumed to be the same size, and placement strategies are typically designed under this uniform assumption. Although this simplification makes analysis easier, it does not reflect many real-world workloads, where input files often vary widely in size. Such heterogeneity can lead to a storage imbalance between workers, an uneven computational load, and inefficient communication during the shuffle phase.

To make CDC applicable in these settings, we extend the framework to nonuniform file sizes. This requires a review of the core design components of CDC. In particular, the file placement strategy must explicitly account for nonuniform file sizes so that storage and computation are balanced across workers.

3.3.1 File Placement Strategy

In the file placement phase, it determines which worker stores which subset of input files and which IVs are locally computable. This decision has a direct and significant impact on the communication load during the Shuffle phase. When input files are of different sizes, they can result in load imbalance and suboptimal performance. Therefore, we adopt a placement strategy that explicitly accounts for nonuniform file sizes.

Let $\mathcal{K} = \{1, 2, \dots, K\}$ be the set of all worker workers and $\mathcal{N} = \{1, 2, \dots, N\}$ denote the set of input files. Each file $n \in \mathcal{N}$ has a file size F_n, and we assume that the files are sorted such that $F_1 \geq F_2 \geq \dots \geq F_N$. When files are of different sizes, they should be treated differently for the file placement strategy. We define a file placement indicator variable $a_{n,S} \in \{0, 1\}$ to denote whether file n is placed exclusively on the subset of workers $S \subseteq \mathcal{K}, S \neq \emptyset$. This allows for the modeling of flexible and potentially overlapping placement strategies. And we have

$$a_{n,\mathcal{S}} \in \{0,1\}, \quad n \in \mathcal{N}, \mathcal{S} \subseteq \mathcal{K} \backslash \emptyset. \tag{3.1}$$

Therefore, to guarantee that each file is placed exclusively in the worker subset, we have the file placement constraint.

$$\sum_{S \subseteq \mathcal{K}, S \neq \emptyset} a_{n,S} = 1, \quad \forall n \in \mathcal{N} \tag{3.2}$$

It ensures that each file is placed exactly once in the system, and it is exclusively assigned to one subset of workers. If we consider a system with $K = \{1,2,3\}$ workers and a single input file, we would denote this file as File 1. We define a binary variable $a_{1,S}$ to indicate whether File 1 is placed in the worker subset $S \subseteq \{1,2,3\}$, where $a_{1,S} = 1$ means that the file is stored in the subset S, and $a_{1,S} = 0$ otherwise.

The values of $a_{1,S}$ for all non-empty subsets S are shown below:

$$\begin{aligned} a_{1,\{1\}} &= 0, & a_{1,\{2\}} &= 0, & a_{1,\{3\}} &= 0, \\ a_{1,\{1,2\}} &= 0, & a_{1,\{1,3\}} &= 1, & a_{1,\{2,3\}} &= 0, \\ a_{1,\{1,2,3\}} &= 0. \end{aligned}$$

This means that File 1 is only stored in the subset $\{1,3\}$. The sum of all placement indicators satisfies the following constraint:

$$\sum_{S \subseteq \{1,2,3\}, S \neq \emptyset} a_{1,S} = 0 + 0 + 0 + 0 + 1 + 0 + 0 = 1.$$

Therefore, the file is placed exactly once in all subsets, which complies with the file placement constraint in the model.

After the file placement strategy, we also need to ensure that the files that are placed in each worker $k \in \mathcal{K}$ do not exceed its mapping load capacity. So we have the following mapping load constraint.

$$\sum_{n=1}^{N} \sum_{\mathcal{S} \subseteq \mathcal{K} \backslash \emptyset, k \in \mathcal{S}} a_{n,\mathcal{S}} F_n \leq m_k, \quad k \in \mathcal{K}. \tag{3.3}$$

3.3.2 Coded Shuffling Strategy

After the file placement is completed and each worker has processed its locally assigned files in the Map phase, the system proceeds to the Shuffle phase. In this phase, each worker needs to acquire the IVs for its assigned target functions that are not computed locally. Therefore, after the Map phase, the system proceeds to the Shuffle phase, where each worker retrieves the missing IVs that were computed by other workers from files it does not store. In particular, those are the IVs for the files not stored by the worker k, given by $\{V_{q,n}, q \in \mathcal{W}_q, n \in \mathcal{N} \backslash \mathcal{M}_k\}$. The coded shuffling phase is conducted for each non-empty worker subset $\mathcal{K} \subseteq \mathcal{K} \backslash \emptyset$. We define $\mathcal{A}_{\mathcal{S}}$ as the set of subfiles stored by the workers in $\mathcal{S}$, which is given by

$$\mathcal{A}_{\mathcal{S}} = \sum_{n=1}^{N} a_{n,\mathcal{S}}. \tag{3.4}$$

If we consider a system with $N = 4$ input files and $K = \{1, 2, 3\}$ workers. The file placement is as follows (only nonzero $a_{n,\mathcal{S}}$ values are listed): File 1 is only stored at Worker 1; Files 2 and 3 are stored at Workers 2 and 3; and File 4 is stored at Workers 1 and 2. Now, we compute the number of files placed in subset $\{2, 3\}$, denoted as $\mathcal{A}_{\{2,3\}}$:

$$\mathcal{A}_{\{2,3\}} = \sum_{n=1}^{4} a_{n,\{2,3\}} = a_{1,\{2,3\}} + a_{2,\{2,3\}} + a_{3,\{2,3\}} + a_{4,\{2,3\}} = 0 + 1 + 1 + 0 = 2.$$

There are two files (File 2 and File 3) that are placed exactly in the subset $\{2, 3\}$. Therefore, $\mathcal{A}_{\{2,3\}} = 2$ indicates that the worker group $\{2, 3\}$ stores two files in total.

Note that the files in $\mathcal{A}_{\mathcal{S}}$ are only available to the users in $\mathcal{S}$. So, only the workers in $\mathcal{S}$ will generate IVs for the files in $\mathcal{A}_\mathcal{S}$, which are given by $\{V_{q,n}, q \in \mathcal{Q}, n \in \mathcal{A}_\mathcal{S}\}$.

For a given non-empty worker subset $\mathcal{S} \subseteq \mathcal{K}$, we define $\mathcal{L}_{k,\mathcal{S}\setminus\{k\}}$ as the set of IVs required by worker k but generated by the other workers in $\mathcal{S} \setminus \{k\}$. Specifically, these IVs are represented by

$$\mathcal{L}_{k,\mathcal{S}\setminus\{k\}} = \left\{V_{q,n} \mid q \in \mathcal{W}_q,\, n \in \mathcal{A}_{\mathcal{S}\setminus\{k\}}\right\},$$

We denote the total number of IVs in this set as $l_{k,\mathcal{S}\setminus\{k\}} \triangleq |\mathcal{L}_{k,\mathcal{S}\setminus\{k\}}|$, which is given by

$$l_{k,\mathcal{S}\setminus\{k\}} = w_q \sum_{n=1}^{N} a_{n,\mathcal{S}},$$

where w_q denotes the number of IVs generated per file.

The IVs in $\mathcal{L}_{k,\mathcal{S}\setminus\{k\}}$ will be delivered to worker k collaboratively by the workers in $\mathcal{S}\setminus\{k\}$. Denote $\mathcal{T}^i_{k,\mathcal{S}\setminus\{k\}} \subseteq \mathcal{L}_{k,\mathcal{S}\setminus\{k\}}$ as the IVs that will be delivered by the worker i for $i \in \mathcal{S}\setminus\{k\}$. To ensure that IVs in $\mathcal{L}_{k,\mathcal{S}\setminus\{k\}}$, those IVs needed by worker k in $\mathcal{S}$ will all be transmitted, we have

$$\bigcup_{i\in\mathcal{S}\setminus\{k\}} \mathcal{T}^i_{k,\mathcal{S}\setminus\{k\}} = \mathcal{L}_{k,\mathcal{S}\setminus\{k\}}, \quad i \in \mathcal{S}, k \in \mathcal{S}, \mathcal{S} \subseteq \mathcal{K}\setminus\emptyset. \tag{3.5}$$

For example, we consider a system with $\mathcal{K} = \{1, 2, 3\}$ workers and $N = 3$ input files. The file placement is as follows: File 1 is stored only at Worker 1. Files 2 and 3 are both stored at Workers 2 and 3.

Assume that Worker 1 is responsible for computing target functions that require the IVs of File 2 and File 3. Since these files are not stored at Worker 1, it must obtain the corresponding IVs from other workers during the Shuffle phase. We define the cooperating worker subset as $\mathcal{S} = \{1, 2, 3\}$. The set of IVs needed by Worker 1 and generated by the other workers in $\mathcal{S} \setminus \{1\} = \{2, 3\}$ is:

$$\mathcal{L}_{1,\{2,3\}} = \{V_{1,2}, V_{1,3}\},$$

where $V_{1,n}$ denotes the IV for file n required by Worker 1.

To ensure that all required IVs are successfully transmitted to Worker 1, for instance, we may assign:

$$\mathcal{T}^2_{1,\{2,3\}} = \{V_{1,2}\}, \quad \mathcal{T}^3_{1,\{2,3\}} = \{V_{1,3}\}.$$

Together, these satisfy the condition required by (3.5)

$$\bigcup_{i\in\{2,3\}} \mathcal{T}^i_{1,\{2,3\}} = \mathcal{L}_{1,\{2,3\}}.$$

This guarantees that all IVs needed by Worker 1 are transmitted and that no IV is missing or duplicated. The collaborative transmission strategy ensures the efficiency of the Shuffle phase.

To quantify the communication load incurred by each worker, we denote $t^i_{k,\mathcal{S}\setminus\{k\}} \triangleq \left|\mathcal{T}^i_{k,\mathcal{S}\setminus\{k\}}\right|$ as the number of IVs delivered by worker $i \in \mathcal{S} \setminus \{k\}$ to worker k. So we can get:

$$\sum_{i\in\mathcal{S}\setminus\{k\}} t^i_{k,\mathcal{S}\setminus\{k\}} = l_{k,\mathcal{S}\setminus\{k\}}, \quad i \in \mathcal{S}, k \in \mathcal{S}, \mathcal{S} \subseteq \mathcal{K}\backslash\emptyset. \tag{3.6}$$

By the definition of $l_{k,\mathcal{S}\setminus\{k\}}$, we have

$$l_{k,\mathcal{S}\setminus\{k\}} = |\mathcal{L}_{k,\mathcal{S}\setminus\{k\}}| = w_q \sum_{n=1}^{N} a_{n,\mathcal{S}},$$

where $w_q = |\mathcal{W}_k|$ denotes the number of target functions assigned to the worker k, and $a_{n,\mathcal{S}}$ indicates whether the file n is stored exclusively in the worker subset $\mathcal{S}$.

Substituting this into Equation (3.6), we obtain the final coded shuffling constraint:

$$\sum_{i\in\mathcal{S}\setminus\{k\}} t^i_{k,\mathcal{S}\setminus\{k\}} = |\mathcal{W}_k| \sum_{n=1}^{N} a_{n,\mathcal{S}}, \quad k \in \mathcal{S}, \mathcal{S} \subseteq \mathcal{K} \setminus \emptyset. \tag{3.7}$$

This equality can make sure that the entire set of required IVs is partitioned and transmitted collaboratively by the workers in $\mathcal{S} \setminus \{k\}$, where

$l_{k,\mathcal{S}\setminus\{k\}}$ is the total number of IVs worker k needs from this group.

This strategy not only ensures correct delivery and flexible sender-side load balancing, but also ensures that no IVs are missed during the shuffle phase. This forms the basis of the coded shuffling constraint, which plays a central role in the efficient implementation of the shuffle phase in heterogeneous coded distributed computing.

Based on the coded shuffling constraint above, each worker $i \in \mathcal{S} \setminus \{k\}$ is responsible for delivering a portion of the intermediate values that the worker needs k. To improve communication efficiency, instead of unicasting each individual IV, the system allows the worker i to multicast a coded message to all other workers in $\mathcal{S} \setminus \{i\}$.

Let $\mathcal{T}^i_{k,\mathcal{S}\setminus\{k\}} \subseteq \mathcal{L}_{k,\mathcal{S}\setminus\{k\}}$ be the subset of IVs assigned to the worker i for delivery to worker k, and define the multicast coded message transmitted by the worker i in the subset $\mathcal{S}$ as:

$$C_{i,\mathcal{S}} = \bigoplus_{k \in \mathcal{S}\setminus\{i\}} \mathcal{T}^i_{k,\mathcal{S}\setminus\{k\}}, \quad i \in \mathcal{S},\ k \in \mathcal{S},\ \mathcal{S} \subseteq \mathcal{K} \setminus \emptyset, \tag{3.8}$$

This coded message is simultaneously useful for multiple workers, as each $\mathcal{T}^i_{k,\mathcal{S}\setminus\{k\}}$ is part of the required IVs of the worker k and is known to all workers in $\mathcal{S} \setminus \{k, i\}$.

which is the XOR of the IVs that all the workers need in $\mathcal{S}\setminus\{i\}$. The size of the coded message $C_{i,\mathcal{S}}$ is

$$|C_{i,\mathcal{S}}| = \max_{k \in \mathcal{S}\setminus\{i\}} t^i_{k,\mathcal{S}\setminus\{k\}}. \tag{3.9}$$

Note that for the special case of $|\mathcal{S}| = 1$, we have only one worker in $\mathcal{S}$, and there are no IVs to be shuffled. For $|\mathcal{S}| = 2$, the worker $i \in \mathcal{S}$ will simply transmit the IVs needed by the other worker in $\mathcal{S}$ without coding.

Through this design, each transmission benefits multiple workers, reducing the total communication load in the Shuffle phase.

3.4 Optimization Formulation and Solution

In the previous section, we described the coded shuffling scheme in detail, where workers transmit IVs required by others using coded messages. Each coded message is constructed to ensure that all required IVs are delivered, thereby reducing the overall communication load.

To further improve system performance, we now aim to minimize the total communication load incurred during the Shuffle phase. This is achieved by jointly optimizing the file placement strategy and the coded shuffling assignments. In what follows, we present a formal optimization problem that captures this objective.

We first define the total shuffled communication load R of the HetCDC scheme as the sum of all coded message sizes $|C_{i,\mathcal{S}}|$, where $\mathcal{S} \subseteq \mathcal{K} \setminus \emptyset$ and $i \in \mathcal{S}$:

$$R = \sum_{i \in \mathcal{S},\ \mathcal{S} \subseteq \mathcal{K} \setminus \emptyset} |C_{i,\mathcal{S}}|. \tag{3.10}$$

Next, following the coded shuffling strategy, each message $C_{i,\mathcal{S}}$ is constructed as a coded combination of IVs intended for the workers in $\mathcal{S} \setminus \{i\}$. Its size is determined by the largest size of IVs that any worker $k \in \mathcal{S} \setminus \{i\}$ requires from worker i,

$$|C_{i,\mathcal{S}}| = \max_{k \in \mathcal{S} \setminus \{i\}} t^{i}_{k,\mathcal{S} \setminus \{k\}}. \tag{3.11}$$

We consider a sender who wants to deliver IVs to two receivers using XOR-based coded multicasting. Receiver A needs IV v_A of size 2 MB. Receiver B needs IV v_B of size 4 MB.

To reduce communication cost, the sender attempts to broadcast a single coded message:

$$C = v_A \oplus v_B.$$

However, the XOR operation requires both operands to have the same length. Therefore, the smaller IV v_A must be padded to match the size of v_B. As a result, the size of the coded message is determined by the largest

IV among the receivers:

$$|C| = \max(|v_A|, |v_B|) = 4\,\text{MB}.$$

This ensures both receivers can correctly decode their desired IVs: Receiver A can compute $v_B = C \oplus v_A$, Receiver B can compute $v_A = C \oplus v_B$.

Thus, the coded message size must be at least the size of the largest IV included in the XOR combination, which explains the use of the max operator in Equation (3.11).

Substituting (3.11) into (3.10) gives the total load:

$$R = \sum_{i \in \mathcal{S},\ \mathcal{S} \subseteq \mathcal{K} \setminus \emptyset} \max_{k \in \mathcal{S} \setminus \{i\}} t^i_{k,\mathcal{S} \setminus \{k\}}. \tag{3.12}$$

With R explicitly defined, the next step is to design a file placement strategy and coded shuffling schedule that jointly minimize this total load. In the following, we formulate the corresponding optimization problem:

$$\mathbf{P0}: \quad \min_{\{a_{n,\mathcal{S}},\, t^i_{k,\mathcal{S}\setminus\{k\}}\}} R$$

subject to:

$$a_{n,\mathcal{S}} \in \{0, 1\}, \quad n \in \mathcal{N}, \mathcal{S} \subseteq \mathcal{K} \setminus \emptyset, \tag{3.1}$$

$$\sum_{S \subseteq \mathcal{K}, S \neq \emptyset} a_{n,S} = 1, \quad \forall n \in \mathcal{N} \tag{3.2}$$

$$\sum_{n=1}^{N} \sum_{\mathcal{S} \subseteq \mathcal{K} \setminus \emptyset, k \in \mathcal{S}} a_{n,\mathcal{S}} F_n \le m_k, \quad k \in \mathcal{K} \tag{3.3}$$

$$\sum_{i \in \mathcal{S} \setminus \{k\}} t^i_{k,\mathcal{S} \setminus \{k\}} = |\mathcal{W}_k| \sum_{n=1}^{N} a_{n,\mathcal{S}}, \quad k \in \mathcal{S}, \mathcal{S} \subseteq \mathcal{K} \setminus \emptyset \tag{3.7}$$

Here, the constraint (3.1) specifies that the file placement indicator $a_{n,\mathcal{S}}$ is binary, indicating whether file n is stored exactly in the set of workers $\mathcal{S}$. Constraint (3.2) ensures that each file is placed exactly once. Constraint (3.3) ensures that the total size of files assigned to any worker k does not exceed its storage capacity m_k. Finally, constraint (3.7) represents the coded shuffling requirement, which ensures that the total number of IVs that worker k needs to send to others matches the number implied by the file placement; here $|\mathcal{W}_k|$ denotes the number of target functions assigned to worker k.

This coupling between the file placement variables $a_{n,\mathcal{S}}$ and the shuffling variables $t^i_{k,\mathcal{S}\setminus\{k\}}$ creates a strong dependency across decision variables, meaning that any change in file placement will affect the required shuffling load for multiple subsets simultaneously. Such interdependence, together with the non-linear max operator in the objective function and the binary nature of the placement variables, makes problem P0 a large-scale integer non-linear program (INLP) problem [96] that is computationally intractable for even moderate system sizes. INLP problems are inherently complex to solve because they may contain multiple local minima and lack closed-form solution structures, which prevents the use of efficient convex optimization techniques. As a result, the problem quickly becomes computationally intractable even for moderate system sizes.

Given these complexities, solving P0 directly becomes impractical for large-scale instances. The combination of discrete decision variables, non-linearities, and the exponential growth of possible file placement patterns with respect to the number of workers K results in prohibitive computational costs when using conventional optimization solvers. To make the problem more tractable while preserving its original structure, it is desirable to reformulate P0 into a form that avoids the non-linear max operator in the objective function.

To address this challenge, we reformulate P0 into an equivalent epigraph form by introducing auxiliary variables $x^i_{k,\mathcal{S}\setminus\{k\}}$ for each $t^i_{k,\mathcal{S}\setminus\{k\}}$ to eliminate the max operations in R, thereby converting the problem into a integer linear programming (ILP) [97]form while preserving the optimality of the original formulation.

$$\mathbf{P1}: \quad \min_{\{a_{n,\mathcal{S}}, t^i_{k,\mathcal{S}\setminus\{k\}}, x^i_{k,\mathcal{S}\setminus\{k\}}\}} \sum_{i\in\mathcal{S},\mathcal{S}\subseteq\mathcal{K}\setminus\emptyset} \sum_{k\in\mathcal{S}\setminus\{i\}} x^i_{k,\mathcal{S}\setminus\{k\}}$$

subject to:

$$a_{n,\mathcal{S}} \in \{0,1\}, \quad n\in\mathcal{N}, \mathcal{S}\subseteq\mathcal{K}\setminus\emptyset, \tag{3.1}$$

$$\sum_{S\subseteq\mathcal{K},S\neq\emptyset} a_{n,S} = 1, \quad \forall n\in\mathcal{N} \tag{3.2}$$

$$\sum_{n=1}^{N} \sum_{\mathcal{S}\subseteq\mathcal{K}\setminus\emptyset, k\in\mathcal{S}} a_{n,\mathcal{S}} F_n \le m_k, \quad k\in\mathcal{K} \tag{3.3}$$

$$\sum_{i\in\mathcal{S}\setminus\{k\}} t^i_{k,\mathcal{S}\setminus\{k\}} = |\mathcal{W}_k| \sum_{n=1}^{N} a_{n,\mathcal{S}}, \quad k\in\mathcal{S}, \mathcal{S}\subseteq\mathcal{K}\setminus\emptyset \tag{3.7}$$

$$t^i_{k,\mathcal{S}\setminus\{k\}} - x^i_{k,\mathcal{S}\setminus\{k\}} \le 0, \quad i\in\mathcal{S},\ k\in\mathcal{S}\setminus\{i\},\ \mathcal{S}\subseteq\mathcal{K}\setminus\emptyset. \tag{3.13}$$

Here, the auxiliary variable $x^i_{k,\mathcal{S}\setminus\{k\}}$ is introduced to the upper bound of each shuffling load variable $t^i_{k,\mathcal{S}\setminus\{k\}}$ in the objective function, thereby replacing the non-linear max operation in R with a set of linear inequality constraints. This transformation preserves the optimality of the original problem while converting it into an ILP formulation, which can be efficiently solved using standard ILP solvers such as Gurobi or Mosek [98] for moderately sized systems. The resulting formulation enables scalable computation by leveraging advanced branch-and-cut [99] and cutting-plane techniques, significantly reducing the computational burden compared to solving the original INLP.

3.5 Numerical Results

To efficiently solve the ILP formulation derived in the previous section, we employ the branch-and-cut method, which is widely recognized as a powerful approach for tackling large-scale integer programming problems. This method combines the branch-and-cut framework with the generation of cutting planes, enabling both systematic exploration of the solution space and effective tightening of the feasible region. In the following subsection, we present the underlying principles of the branch-and-cut method and the numerical results obtained from our experiments.

The branch-and-cut method is an advanced integer programming technique that integrates the branch-and-bound framework with the generation of cutting planes. It iteratively partitions the solution space into smaller sub-problems (branching) while tightening the feasible region through valid inequalities (cutting), aiming to reduce the search space without excluding any feasible integer solutions [99].

The complexity of the branch-and-cut method comes from multiple factors. First, the search tree may grow exponentially with the size of the problem, especially when the integrality gap is large. Second, the generation and selection of effective cutting planes is itself a combinatorial problem that can significantly impact convergence speed. Finally, in large-scale optimization, the interaction between branching rules and cut selection introduces additional computational overhead, requiring careful balancing between exploration and pruning efficiency.

In our implementation, the reformulated ILP model in (3.1) was solved using the commercial solver Mosek, which incorporates a highly optimized branch-and-cut engine. The solver begins by relaxing the binary placement variables to continuous values in $[0, 1]$ and solving the LP relaxation to establish a lower bound. When the LP relaxation yields fractional solutions, Mosek selects a branching variable to partition the problem into two subproblems, each fixing the variable to 0 or 1. Concurrently, the solver performs cut separation to identify and add violated valid inequalities, thereby tightening the feasible region and improving the bound quality.

To accelerate convergence, Mosek applies advanced cutting-plane strategies in combination with presolve procedures that streamline the problem structure, thereby reducing complexity and improving computational efficiency. The branch-and-cut process terminates when the relative optimality gap between the best known feasible solution and the lower bound falls below the specified tolerance ε_{gap}. For our experiments, the tolerance was set to 10^{-6}, with multi-threading enabled to utilize available computational resources fully.

This configuration enabled the solver to find globally optimal solutions for moderately sized problem instances within seconds to minutes, significantly reducing the computational burden compared to directly solving the original non-linear problem. With the optimized solutions obtained from the ILP formulation, we proceed to evaluate the performance of the proposed HetCDC scheme through numerical experiments.

Due to constraint (3.1), the problem **P1** can be formulated as an integer linear program (ILP), which is solved using the branch-and-cut method implemented in the commercial optimization solver MOSEK. By applying the aforementioned reformulation and solution strategy, the optimized HetCDC scheme is obtained. We evaluated the optimized HetCDC proposed against two baselines: the homogeneous CDC (HomCDC) scheme [24] and the nested CDC (N-CDC) scheme [100], under heterogeneous worker capacities and nonuniform file sizes. All methods are implemented by solving the ILP formulation with a commercial solver (MOSEK) using the branch-and-cut method.

In Fig. 3.5, we plot the total communication load R versus the number of input files N for a MapReduce system with $K = 8$ workers. To capture worker heterogeneity, we adopt a nonuniform mapping load profile $\mathbf{m} = [2, 3, 3, 4, 5, 5, 6, 7]$, and file sizes are set as $F_n = 1/n$ Kbits. The results show that the optimized HetCDC consistently achieves the lowest shuffling load among all the schemes considered, confirming its effectiveness in handling nonuniform file sizes. We also observe that the performance gap between HetCDC and both HomCDC and N-CDC narrows when N is small. This indicates that HetCDC is more sensitive to file placement opportunities at low file granularity, while its advantage becomes more pronounced as N increases.

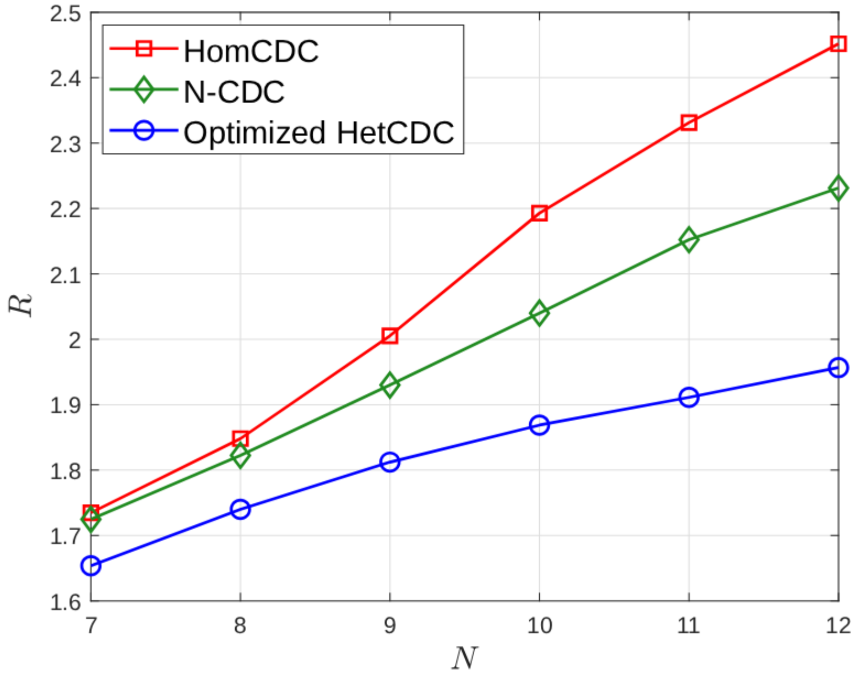


Fig. 3.5: Communication load R (normlized by file size) vs. number of files $N(K = 8$ workers with file sizes $F_n = 1/n$ Kbits and mapping-load vector $\mathbf{m} = [2, 3, 3, 4, 5, 5, 6, 7]$).

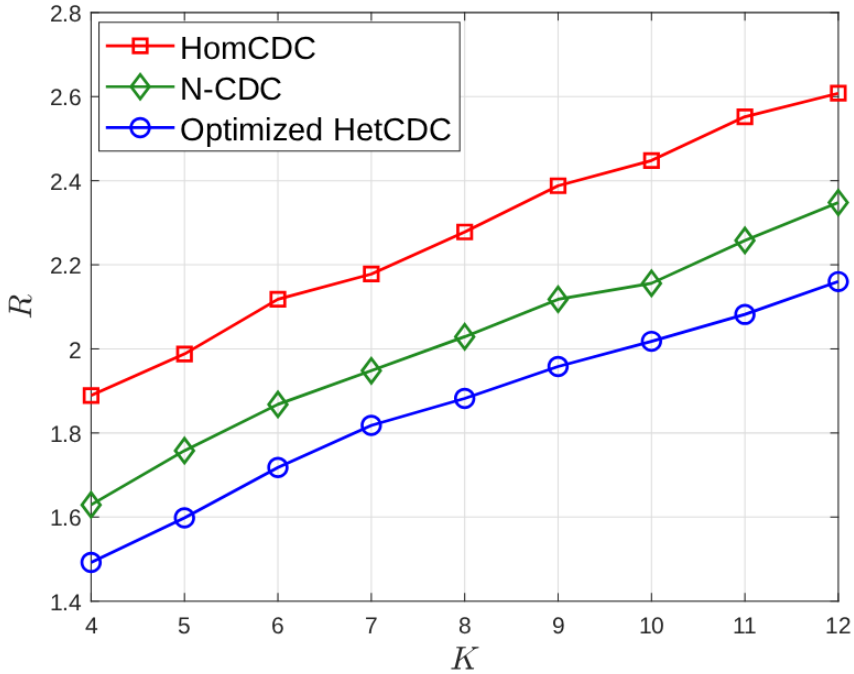


Fig. 3.6: Communication load R (normlized by file size) vs. number of workers $K(N =$ 12 files with file sizes $F_n = 1/n$ Kbits and mapping load $m_k = \frac{k}{K(K+1)/2}$ for all k).

In Fig. 3.6, we plot R versus the number of workers K with $N = 12$ files of size $F_n = 1/n$ Kbits. The mapping load is set as $m_k = \frac{k}{K(K+1)/2}$ for all $k \in \mathcal{K}$ to simulate the heterogeneous system. The optimized HetCDC again yields the lowest communication load for all tested cluster sizes. As the number of workers grows, the coding opportunities enabled by heterogeneity-aware file placement become more significant, whereas the other schemes do not fully exploit such opportunities.

The numerical results demonstrate that the proposed HetCDC consistently outperforms both HomCDC and N-CDC under heterogeneous settings, highlighting the importance of explicitly input file size heterogeneity. When the number of files N is small, the performance gap between HetCDC and the baselines is limited because fewer opportunities restrict coding flexibility; however, as N increases, nonuniform file sizes provide richer placement options, leading to more coding opportunities and larger reductions in shuffle load. Similarly, when scaling the number of workers K, HetCDC shows superior scalability since it optimizes placement and shuffling under heterogeneous worker capacities, while HomCDC and N-CDC remain constrained by uniform assumptions that cause imbalance and inefficiency. Overall, these results confirm that under heterogeneous conditions, schemes such as HomCDC that ignore file-size heterogeneity, or schemes like N-CDC that only partially account for it, lead to suboptimal performance, whereas HetCDC achieves consistent communication savings across all scales considered. And in next section, we will apply HetCDC in TeraSort.

Chapter 4

Heterogeneous Coded TeraSort

TeraSort is a widely adopted benchmark for evaluating the performance of large-scale distributed sorting systems[25]. Due to the massive volume of data exchanged during the Shuffle phase, it serves as an ideal workload for studying communication-efficient strategies. Existing work, such as Coded TeraSort[26], has demonstrated the benefits of applying coding techniques to reduce the communication load. However, most of these approaches assume uniform input file sizes and homogeneous system configurations. In this chapter, we first review the Coded TeraSort algorithm to highlight its core ideas and limitations. In this chapter, we introduce a new Heterogeneous Coded TeraSort algorithm that integrates the HetCDC framework proposed in Chapter 3 into the TeraSort algorithm. By adapting file placement and coded shuffling strategies to handle nonuniform file sizes and heterogeneous worker capabilities, our method achieves improved communication efficiency in realistic distributed sorting scenarios.

4.1 Preliminary

The Coded TeraSort is an extension of the TeraSort algorithm[26]. It applies an idea from coded distributed computing to reduce the communication load during the shuffle phase. The key idea is to exploit the redundant file placement strategy across multiple workers, enabling the system to create coded messages that simultaneously satisfy their data needs. Instead, each IV is sent directly to the requesting worker. Workers combine several IVs into a single coded message using bitwise XOR operations. These coded transmissions are carefully designed so that each recipient can decode their required files using the local files. The implementation consists of five components: Redundant File Placement, Map Phase, Multicasting Shuffle Phase, Decoding Phase, and Reduce Phase.

First, the input KV pairs are split into many small files, and each of them is placed on workers repeatedly based on a special strategy. Then the operation is the same as that with the TeraSort map phase operation. In the multicasting shuffle phase, each worker generates the IV from the map phase. Each worker needs to multicast its IV to a specific set of other workers. Next, each worker decodes the needed KV pairs from the received coded message. Finally, we have the reduce phase; each worker sorts the KV pairs within its partition. Next, we will describe the above 5 phases in detail. We have the same example with TeraSort to illustrate these phases.

The first phase is the redundant file placement phase. Now we consider about $K = 4$ workers and input files $\mathcal{N} = \{F_{1,2}, F_{1,3}, F_{1,4}, F_{2,3}, F_{2,4}, F_{3,4}\}$ repetitively on $r = 2$ different workers. As shown in Figure 4.1, each file $F_{i,j}$ is stored in workers i and j. For example, the file $F_{1,2}$ is stored at worker 1 and worker 2, while $F_{2,4}$ is stored at worker 2 and worker 4. Each file consists of multiple key-value pairs that are to be processed in the Map phase.

In the map phase, each file is stored redundantly on two workers, and each worker stores three files. As shown in the figure 4.1, Worker 1 obtains files $F_{\{1,2\}}, F_{\{1,3\}}$, and $F_{\{1,4\}}$. During the map phase, Worker 1 processes each file to calculate IVs. For example, file $F_{\{1,2\}}$ contains $\{1, 26, 52, 75\}$, and Worker 1 hashes and generates IVs for each key. The IVs are represented according to their key and files, e.g., $I^1_{\{1,2\}} = [1]$, where the superscript 1 indicates partition 1. Similarly, Worker 1 computes IVs from files $F_{\{1,3\}}$, $F_{\{1,2\}}$ and $F_{\{1,4\}}$. The resulting IVs are grouped and color-coded based on their corresponding target keys and partitions. Specifically, red corresponds to the range $[0, 25)$, blue to $[25, 50)$, green to $[50, 75)$, and purple to $[75, 100)$.

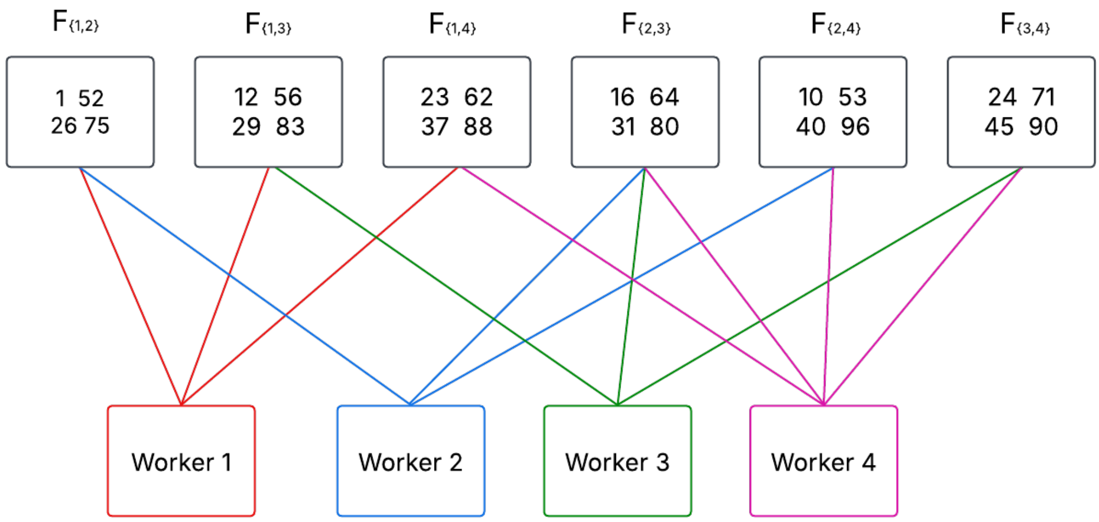


Fig. 4.1: Illustration of file placement in Coded TeraSort for $K = 4$ with $r = 2$.

As we know, in the Map phase, each worker processes its locally stored files and generates IVs for all key partitions. Each IV is labeled according to the key's destination partition and the origin of the file. Due to file redundancy, a known IV may be computed in multiple workers.

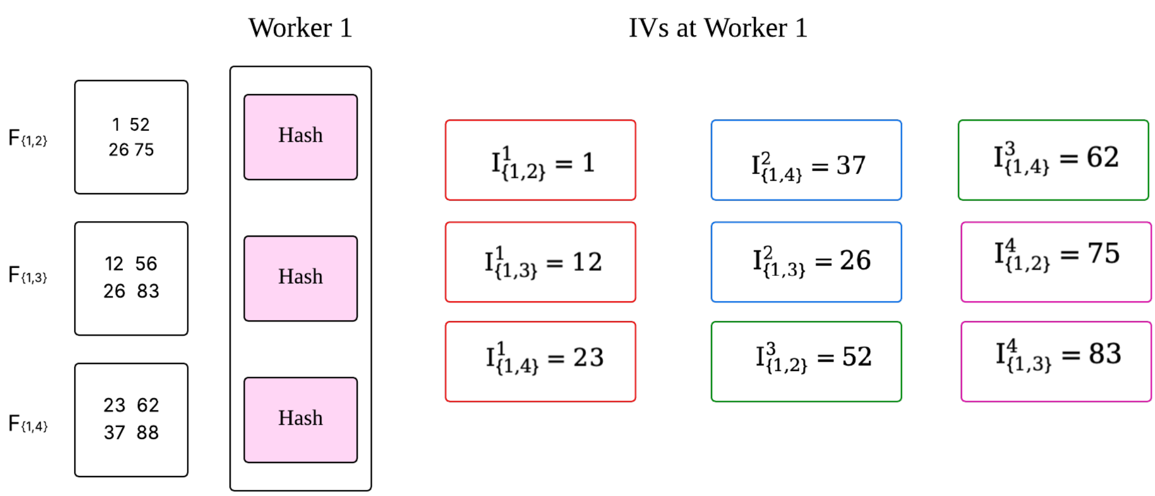


Fig. 4.2: Illustration of Coded Terasort in Map phase for $K = 4$ with $r = 2$ and key partitions[0,25),[25,50),[50,75),[75,100).

In the Shuffle phase, instead of sending IVs individually to their respective destinations, workers combine multiple IVs into a single coded multicast message using bitwise XOR. For example, shown in Figure 4.3, where a coded message of the form $I_a \oplus I_b$ is multicast to two workers: one can decode I_a, the other can decode I_b, using their own locally computed IVs.

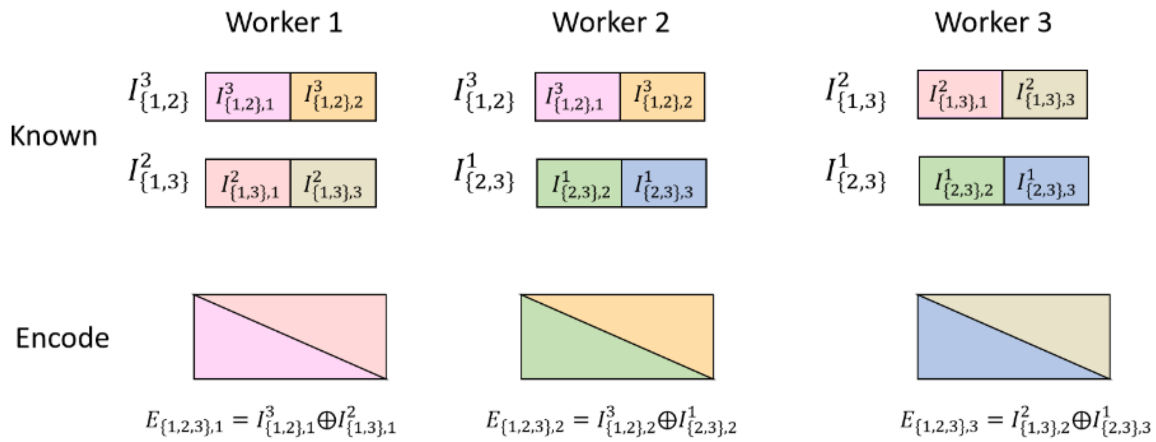


Fig. 4.3: An example of Coded Terasort encoding process.

In the decoding phase, each worker decodes the needed IVs using the received coded message and the IVs it already obtains locally. The decoding is enabled by the XOR operation. For example, shown in Figure 4.4 worker 1 receives $E_{\{1,2,3\},1} = I^3_{\{1,2\},1} \oplus I^1_{\{1,3\},1}$, and since it has $I^1_{\{1,3\},1}$, it can decode:

$$I^3_{\{1,2\},1} = E_{\{1,2,3\},1} \oplus I^1_{\{1,3\},1}$$

In the reduce phase, once all required KV pairs have been obtained, either from local computation or through decoding, each worker proceeds to the final step: sorting its assigned data. Specifically, the worker k sorts all the KV pairs belonging to its partition P_k, ensuring that the output is globally ordered when all partitions are combined. And this step is the same as for the classical Terasort.

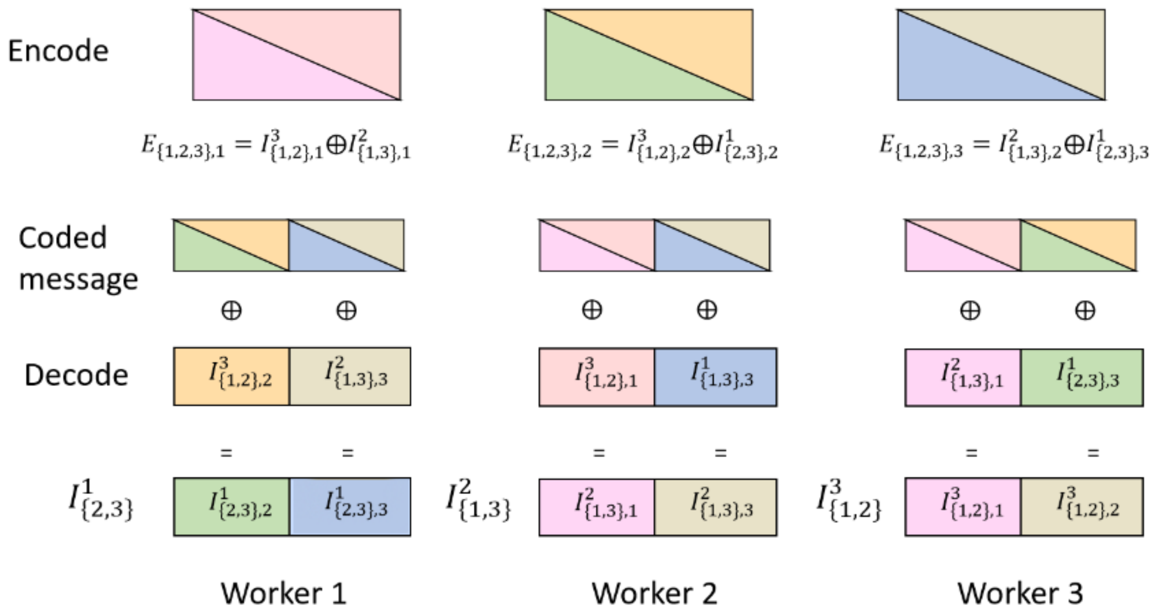


Fig. 4.4: An example of decoding process.

4.2 Heterogeneous Coded TeraSort with Nonuniform File Sizes

Although Coded TeraSort significantly reduces the communication load in the shuffle phase, its design relies on the simplifying assumption that all input files are of uniform size and does not explicitly account for per-worker resource limits. However, real-world datasets are also composed of files with highly nonuniform sizes; such heterogeneity leads to unbalanced storage and computing pressure across workers and limits the effectiveness of homogeneous coded schemes. To address these issues, we extend the HetCDC framework proposed in Chapter 3 to the TeraSort and develop the Heterogeneous Coded TeraSort algorithm under nonuniform file sizes.

The difficulty arises from two kinds of heterogeneity. First, the input KV pair files have nonuniform sizes. Under structured redundant file placement with replication level r, a naive placement strategy can easily overload a few workers while leaving others lightly used, or exceed worker capacity limits, even though the subset pattern required for coding is satisfied. Such an imbalanced placement will lead to the Map phase. Workers assigned large files create much larger volumes of IVs, delaying their local processing and making them the bottleneck in Shuffle.

Second, even with a valid placement, the hashing operation, as in TeraSort, routes uneven amounts of KV pairs from a given file to different partitions. Standard homogeneous coded designs implicitly rely on pairing equal amounts across receivers, but unequal input files make many intended coding opportunities unworkable at a given subset size.

Correctness imposes additional constraints on the Shuffle strategy. Coded packets must be formed by XOR over equal-length chunks so that every receiver can cancel the parts they don't need using only its locally available side information created by replication. For every KV pair, all IVs required at W_q must be recoverable from the received coded packets together with local side information so that $P_1, \ldots, P_K$ partitions can generate the correct globally sorted output.

In summary, the problem is choosing a structured redundant file placement that respects the limits of the per-worker capacity while avoiding severe imbalance, and an operating encoding and multicast shuffling strategy remains decodable under different input file sizes, to minimize the Shuffle communication load while keeping the Map and Reduce functions the same as Coded TeraSort.

System Model

We study the Coded TeraSort with heterogeneous inputs and worker capacity. Consider K workers $\mathcal{W} = \{W_1, \ldots, W_K\}$ and K reduce partitions $\mathcal{P} = \{P_1, \ldots, P_K\}$, where W_k outputs the final partition P_k. The input KV pairs $(key_i, value_i)$ are split into N files $F = \{F_1, \ldots, F_N\}$ with nonuniform sizes $|F_n|$ (bytes). Workers may have heterogeneous storage capacities B_k. We adopt a replication level $r \in \{1, \ldots, K\}$ and place each file F_n on exactly r workers. All workers use the same hash partitioner that maps each key to an index in $[K]$. We index reducers and partitions by $q \in [K]$, so W_q outputs P_q.

For later reference, let $I_k^{(n),q}$ denote the IV produced by worker W_k from file F_n that is destined for partition P_q. During the shuffle phase, workers exploit the replica structure to multicast coded transmissions to the appropriate reducers, using simple length alignment (zero padding within a subset) when needed. Finally, each W_q receives its assigned records, performs local sort and outputs P_q; concatenating $P_1, \ldots, P_K$ partitions the globally sorted result. The system model is illustrated in Fig. 4.5.

The remainder of this section specifies the workflow under this setting: we first introduce the file placement algorithm according to nonuniform sizes, then recall the Map operation, describe the coded Shuffle with length alignment and its decoding, and finally summarize the Reduce phase.

File Placement Algorithm

We adopt a file placement algorithm that achieves heterogeneous storage capacity and prepares the data for efficient coded shuffling. The design targets need to achieve two goals: (i) balancing byte volume across workers to avoid workers mapping load, and (ii) improving coding efficiency in the shuffle phase by keeping within each replica subset, the lengths of IVs that target the same partition close, so that length alignment by zero padding is small.

To realize the two goals, we use a grouped replication scheme driven by the replication level r. Since the system stores exactly r replicas per file, the total replica count is rN; thus, a natural target per worker is rN/K files. We first order files by decreasing size using First-Fit Decreasing (FFD) [101] and partition this ordered list into a few scale tiers: a large-file tier $\mathcal{L}$, a small-file tier $\mathcal{S}$, and (optionally) an intermediate tier $\mathcal{M}$ when needed to match capacities.

Replication then proceeds in tiers: files in $\mathcal{L}$ are replicated only between a designated large block of workers, files in $\mathcal{S}$ only between a small block and files in $\mathcal{M}$ among the remaining middle workers; within each block and for each file, we choose r different workers by selecting those with the largest residual byte capacity (and, secondarily, the largest gap to the target file per worker) so that no worker exceeds B_k and each worker approaches the rN/K files.

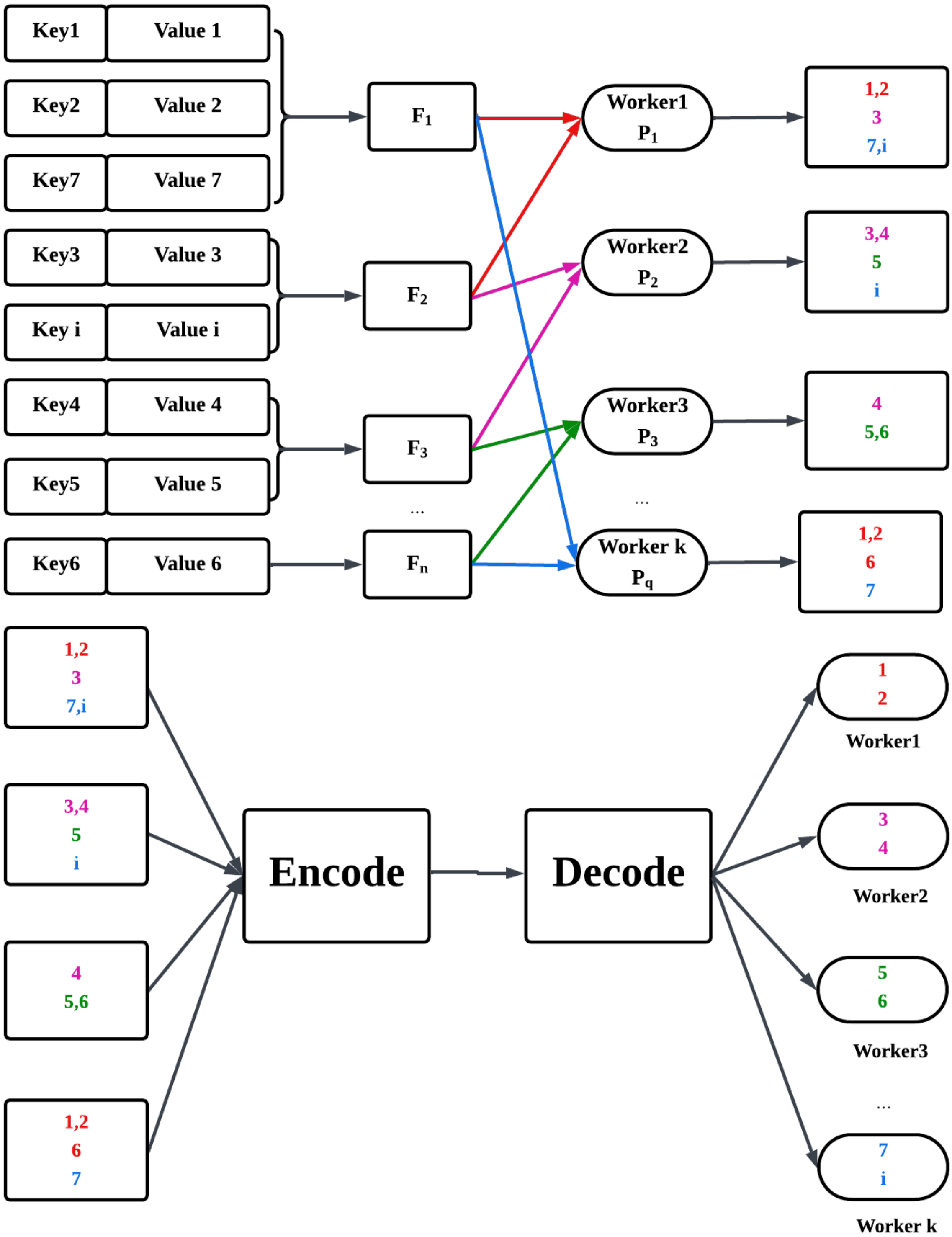


Fig. 4.5: Heterogeneous Coded TeraSort System model.

This "large-with-large, small-with-small" achievement preserves full nonuniformity, no equalization, splitting, or merging, while ensuring overlap: any two workers within the same block share many common replicas, so the assigned files of each worker have redundancy in other workers by construction. Because the map phase uses a common hash partitioner, files of comparable scale yield intermediate components of comparable length toward each reducer; So coded multicasting formed within a block requires only small zero padding to align lengths, improving shuffle efficiency without changing original file sizes.

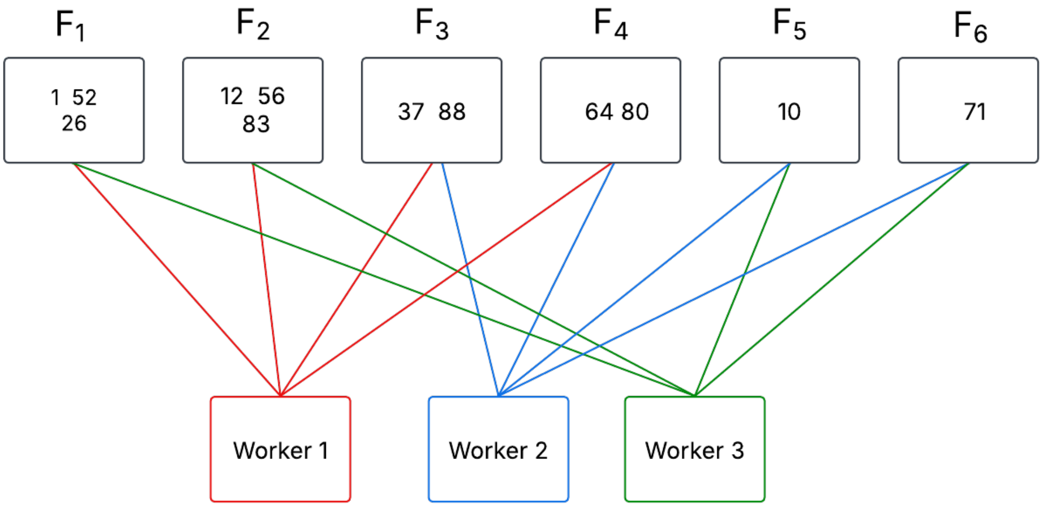


Fig. 4.6: Illustration of file placement in Heterogeneous Coded TeraSort for $K = 3$ with $r = 2$.

As an illustration with $K = 3$, Figure 4.6 shows six input files $F_1, \ldots, F_6$ and the workers that store them. The colored edges denote placement (red$\rightarrow W_1$, blue$\rightarrow W_2$, green$\rightarrow W_3$). The keys are routed by three partitions.

$$\mathcal{P}_1 = [1, 33], \qquad \mathcal{P}_2 = (33, 66], \qquad \mathcal{P}_3 = (66, 100],$$

In this example, the replication level varies due to nonuniform sizes and capacity limits:

$$\begin{array}{lll} F_1 \text{ on } \{W_1, W_3\}, & F_2 \text{ on } \{W_1, W_3\}, & F_3 \text{ on } \{W_1, W_2\}, \\ F_4 \text{ on } \{W_1, W_2\}, & F_5 \text{ on } \{W_2, W_3\}, & F_6 \text{ on } \{W_2, W_3\}. \end{array}$$

This structured distribution creates shared side information within each holder subset (e.g., $\{W_1, W_3\}$ for F_1). The partitions $\mathcal{P}_1$–$\mathcal{P}_3$ are then used during shuffle to route emitted records to reducers.

Map

Each worker W_k scans its local files and applies the common hash partitioner. For a local file F_n at W_k, the map step parses its KV records and routes them by $q = h(\text{key})$ into K destination streams. We denote by $I_k^{(n),q}$ the collection of IVs produced at W_k from F_n whose keys map to destination q; equivalently, these values are destined for reducer W_q and belong to partition P_q.

Before entering the shuffle phase, the KV pairs will preserve the record order within (n, q). Every KV pair contains lightweight metadata (n, q, value) so that reducers can verify partition membership and concatenate in order. We do not equalize, split, or merge original input files. In particular, the heterogeneity of $\{|F_n|\}$ is preserved. After the map phase, the worker exposes exactly the information needed by the subsequent coded Shuffle.

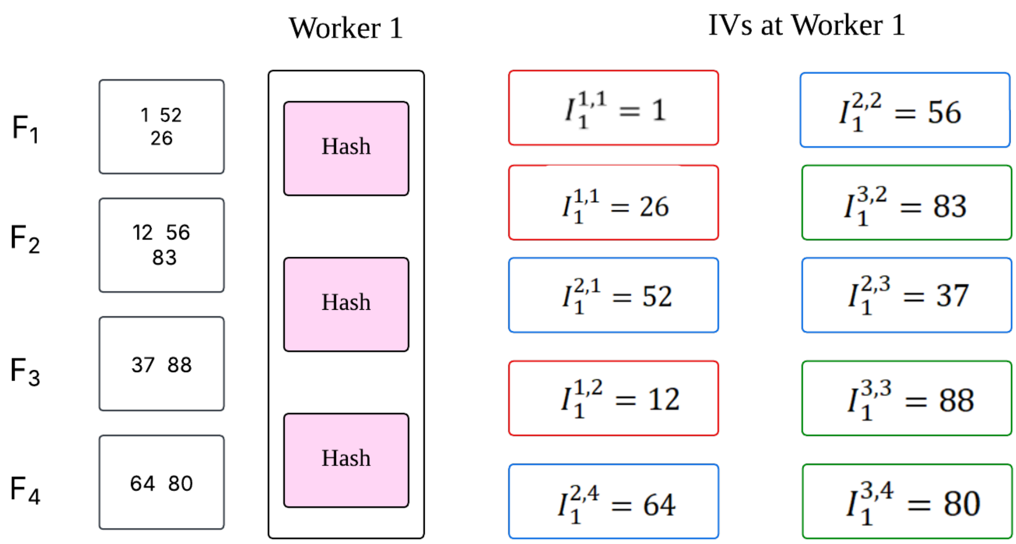


Fig. 4.7: Illustration of Heterogeneous Coded Terasort Map phase for $K = 3$ with $r = 2$ and key partitions(0,33],(33,66],(66,100].

As an illustration in Fig. 4.7, Worker 1 maps its locally stored files F_1–F_4 and hashes each key to one of the three partitions. From the sample keys shown in the figure, Worker 1 produces:

$$\begin{aligned}
&\mathcal{P}_1 : I_1^{(1),1}(1),\ I_1^{(1),1}(26),\ I_1^{(2),1}(12);\\
&\mathcal{P}_2 : I_1^{(1),2}(52),\ I_1^{(2),2}(56),\ I_1^{(3),2}(37),\ I_1^{(4),2}(64);\\
&\mathcal{P}_3 : I_1^{(2),3}(83),\ I_1^{(3),3}(88),\ I_1^{(4),3}(80).
\end{aligned}$$

Here, the number in parentheses is a representative key value falling in P_q. Worker 1 retains all IVs for its own reducer (P_1) and sends the IVs to other partitions (P_2, P_3) for the Shuffle phase. This results in redundant side information at holders of the duplicate files, which we will exploit via length-aligned coded multicasting during shuffle.

Shuffle

For each partition index q and each file F_n, reducer W_q must obtain the IVs $\{I_k^{(n),q} : k \in \mathcal{S}_n\}$ produced at the r holders of F_n. Delivery is performed inside the holder set $\mathcal{S}_n$ using XOR and local length alignment. When combining several IVs in one XOR, operands are zero–padded only up to the maximum length among the operands in that XOR. So, the heterogeneity of $\{|F_n|\}$ is preserved.

Fix a pair (n, q) and a transmitter $t \in \mathcal{S}_n$. The worker W_t transmits one packet

$$X_t^{(n),q} = \bigoplus_{u \in \mathcal{S}_n \setminus \{t\}} \mathrm{pad}\Big(I_u^{(n),q}[\tau_{t,u}]\Big),$$

where $\tau_{t,u}$ selects the next unsent IVs needed by receiver u, and pad$(\cdot)$ zero–padding to the longest operand within this packet. As t ranges over $\mathcal{S}_n$, the indices $\{\tau_{t,u}\}$ are advanced so that every piece of each $I_u^{(n),q}$ is covered exactly once. For a packet sent by W_t, each receiver $u \in \mathcal{S}_n \setminus \{t\}$ can reproduce all XOR operands except its own target piece $I_u^{(n),q}[\tau_{t,u}]$ using its local replicas and the common hash partitioner from the map phase. Therefore, u cancels the known terms and recovers the missing piece. Repeating over $t \in \mathcal{S}_n$ produces all the pieces for (n, q).

Packets are formed so that operands combined in the same XOR have comparable lengths (a direct consequence of placing large files with large files and small with small), which keeps padding small. After all required pieces are decoded, W_q concatenates them in (n, q, t) order and proceeds to reduce phase.

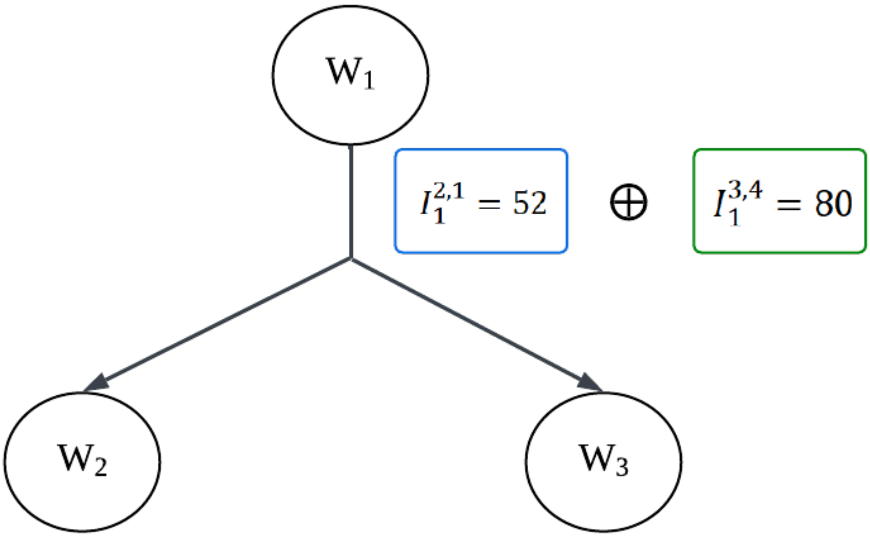


Fig. 4.8: Illustration of Heterogeneous Coded Terasort in shuffle phase for Worker 1.

In Fig. 4.8, W_1 serves $\mathcal{S} = \{W_2, W_3\}$ by pairing two IVs: $I_3^{(1),2}$ (blue; destined to P_2 from F_1) and $I_2^{(4),3}$ (green; destined to P_3 from F_4). Due to placement, W_2 already has $I_2^{(4),3}$ and W_3 already has $I_3^{(1),2}$. W_1 broadcasts the same encoded payload to both receivers:

$$X = I_3^{(1),2} \oplus I_2^{(4),3}.$$

In Fig. 4.8, worker W_1 serves the subset $\mathcal{S} = \{W_2, W_3\}$ by pairing two IVs needed by the receivers: $I_3^{(1),2}$ (blue; produced at W_3 from F_1, destined for P_2) and $I_2^{(4),3}$ (green; produced at W_2 from F_4, destined to P_3). By placement, W_2 already holds $I_2^{(4),3}$ and W_3 already holds $I_3^{(1),2}$. W_1 broadcasts a single coded packet to both receivers, with local length alignment to the longer operand:

$$X = \mathrm{pad}\big(I_3^{(1),2}, L\big) \oplus \mathrm{pad}\big(I_2^{(4),3}, L\big), \qquad L \triangleq \max\Big\{\big|I_3^{(1),2}\big|, \ \big|I_2^{(4),3}\big|\Big\}.$$

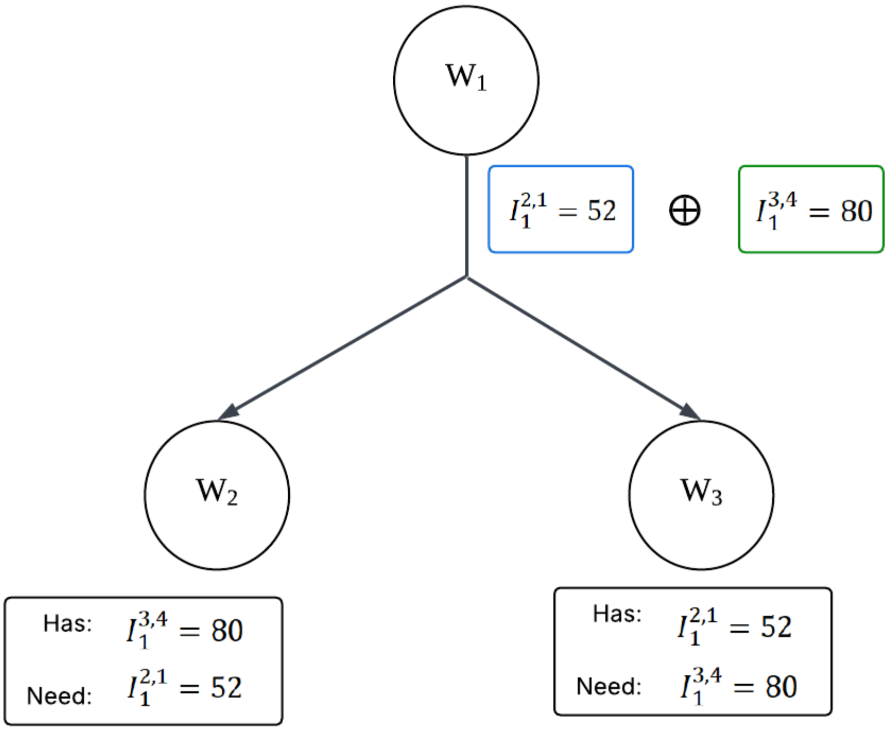


Fig. 4.9: Illustration of Heterogeneous Coded Terasort for decoding.

At W_2: using its side information $I_2^{(4),3}$, it cancels and recovers the missing IV for partition P_2:

$$I_3^{(1),2} = X \oplus \mathrm{pad}\big(I_2^{(4),3}, L\big).$$

At W_3: symmetrically it recovers the IV for partition P_3:

$$I_2^{(4),3} = X \oplus \mathrm{pad}\big(I_3^{(1),2}, L\big).$$

If the two IVs already have the same length, pad$(\cdot)$ is a no operation and the packet reduces to $X = I_3^{(1),2} \oplus I_2^{(4),3}$.

Reduce

After decoding, reducer W_q has obtained all KV pairs whose keys are assigned to partition P_q. The reducer W_q then: (i) reassembles each sequence by concatenating the received pieces in (n, q, t) order to form the full byte ranges $I_k^{(n),q}$; and (ii) performs an external merge sort by key over the union producing the final output P_q written by W_q.

Because the map function assigns each key to exactly one index q and the shuffle phase recovers all $I_k^{(n),q}$ losslessly, P_q contains precisely the KV pairs for partition q without loss and duplication. When the partitioner is in the total range, concatenating $P_1, \ldots, P_K$ generates a globally sorted output; Under a hash partitioner, each P_q is internally sorted by key range.

4.3 Experimental Results

We evaluated the proposed Heterogeneous Coded TeraSort (HCT) by modifying an open-source OpenMPI(C++) implementation of Coded TeraSort. The implementation follows the standard design where each input record is a 100-byte key–value (KV) pair with a 10-byte unsigned integer key and a 90-byte value, and the records are ordered by the integer key. We keep this data format unchanged in our modifications.

Cluster and software

Experiments run on a controlled cluster with K workers (VMware VMs, each with 2 GB memory and a single-core 3 GHz CPU). The inter-worker link is rate-limited to 100 Mbps to emulate a modest datacenter bandwidth. The codebase is compiled with g++ and executed with OpenMPI. In the code, the number of reducers is set as $numReducer = K$. For the coded version, the computation load (replication level) is set by $load = r$; the implementation expects $numInput = K$ input partitions, consistent with the reference design. We invoke the pipeline via ./Splitter followed by mpirun -np (K) to spawn K workers plus one master.

Dataset and heterogeneity

The input is generated by TeraGen in the standard TeraSort format, producing 2 GB of 100-byte KV pairs (10-byte key, 90-byte value). To emulate heavy-tailed file sizes observed in practice, we partition the 2 GB input into three size classes and form files of approximately 110 MB (large), 15 MB (medium), and 2 MB (small).

Placement and coding in our system

Relative to the reference implementation, we enable heterogeneous file sizes and apply a size-aware placement algorithm: files are first sorted by size and grouped within the same size group before coded shuffling. This reduces zero-padding when forming XORs of IVs under nonuniform inputs. We keep the standard TeraSort partitioner with K key ranges, and vary r via *load* to compare uncoded ($r = 1$) and coded runs ($r \geq 2$).

Figure 4.10 shows the shuffle load R with the number of workers K. All three methods incur larger communication when K grows, but the increase is the highest for the uncoded baseline, milder for the homogeneous coded baseline, and slowest for our heterogeneous coded design. At $K = 10$ we observe roughly $R \approx 7.2$ (uncoded), $R \approx 5.6$ (homogeneous coded), and $R \approx 4.8$ (proposed). The gap widens as K increases, indicating that size-aware grouping unlocks more efficient multicasts on a larger scale.

Figure 4.11 varies the computation load r on a fixed K. As expected, the uncoded baseline remains essentially flat, whereas both coded schemes monotonically reduce R when r increases. Across the full range, the proposed method dominates the homogeneous coded baseline. At small to moderate replication (for example, $r = 2$ -5), the advantage is more pronounced ($R \approx 4.1$ vs. 5.4 at $r = 2$; $R \approx 2.0$ vs. 3.6 at $r = 5$).

These trends highlight a trade-off between computing and communication load: a larger r increases map and memory but decreases shuffle cost. Because our method attains a given R at a smaller r than the homogeneous coded baseline, users can finish a target shuffle phase with lower replication (and thus lower CPU/memory overhead). The size-aware placement first sorts files and groups them within the same size class before coding. This alignment reduces zero padding inside the XORs. It improves the packing efficiency of each coded transmission, which explains (i) the slower growth of R with K in Figure 4.10 and (ii) the faster decrease of R with r in

Figure 4.11.

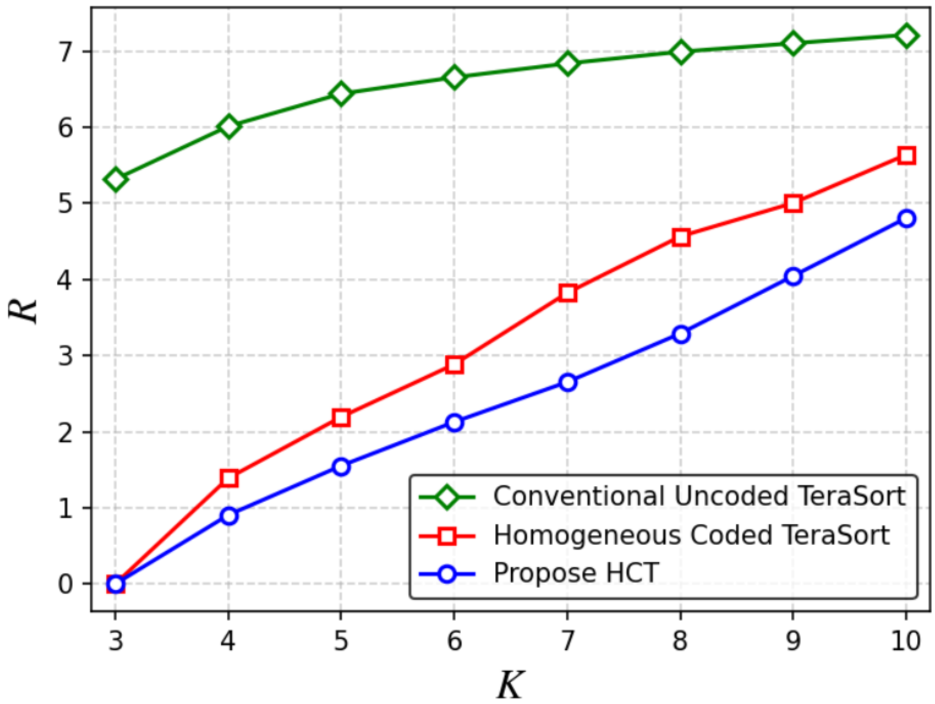


Fig. 4.10: Shuffle load R (normlized by file size) vs. number of workers K.

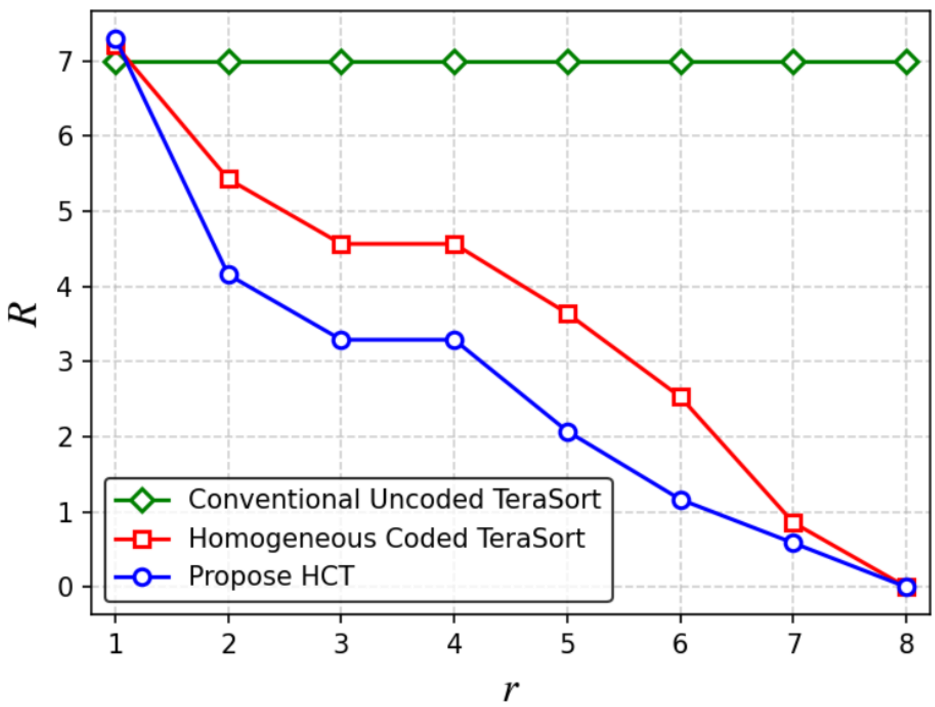


Fig. 4.11: Shuffle load R (normlized by file size) vs. replication level r.

Summary

Across all experiments, the proposed HCT achieves the lowest shuffle load R among the three methods. The advantage increases with K and, for a fixed K, decreases more rapidly with r, reaching a target R at a smaller replication level than the homogeneous coded baseline. These gains come from size-aware file placement that aligns file sizes before coding, which reduces padding and improves multicast packing efficiency. Practically, a modest r suffices to satisfy shuffle budgets, reduces CPU memory overhead, highlighting the benefit of a heterogeneity-aware design.

Chapter 5

Conclusion and Future Work

In this thesis, we investigate the limitations of conventional CDC, which assumes uniform file sizes and homogeneous worker capabilities. Such assumptions, while analytically convenient, significantly constrain the applicability of CDC in practical large-scale systems where heterogeneity is inevitable. To overcome these limitations, we proposed a HetCDC framework that explicitly accounts for nonuniform file sizes and heterogeneous worker capacities. By jointly optimizing the file placement and coded shuffling strategies, the proposed scheme can substantially reduce the communication load during the shuffle phase.

To address the computational challenges of the optimization problem, which is inherently non-convex and non-linear, we reformulated it into an ILP problem through the introduction of auxiliary variables. The reformulated problem was then solved using the branch-and-cut method, which enables scalable and efficient optimization. The numerical results confirmed that HetCDC consistently achieves a lower communication load compared with HomCDC and N-CDC. These results validate the effectiveness of considering file size heterogeneity in the design of coded distributed computing frameworks.

Building on the HetCDC framework, we develop a Heterogeneous Coded TeraSort (HCT) that combines size-aware file placement with coded multicasting in the Map and Shuffle phase. The approach consistently lowers the normalized shuffle load R compared with both the uncoded design and the homogeneous coded baseline. For a fixed number of workers K, increasing the replication level r monotonically decreases R, with HCT delivering the largest reductions, whereas the uncoded scheme remains essentially flat. As the cluster size K grows, R increases for all methods, but HCT exhibits the slowest growth and preserves a clear gap to the baselines. These reductions in R translate into shorter shuffle time and improved end-to-end sorting performance in our experiments.

In the future, several promising research directions can be pursued. First, while this work has focused on small sized clusters, further investigation is required to extend HetCDC to large-scale and high-dimensional environments, where additional scalability challenges may arise. Second, it would be beneficial to explore hybrid optimization approaches, such as heuristic algorithms, approximation schemes, or distributed solvers, to reduce computational complexity and enable real-time adaptation in dynamic systems. Third, the integration of HetCDC with emerging paradigms such as federated learning, edge computing, and serverless architectures presents exciting opportunities for cross-domain applications. Finally, deploying and validating the HetCDC framework on cloud platforms such as Amazon EC2 or Microsoft Azure will provide further insights into its robustness, scalability, and cost efficiency in practical heterogeneous settings.

In summary, this thesis established a comprehensive framework for heterogeneous coded distributed computing, provided optimization techniques, and demonstrated its applicability through both numerical analysis and system-level implementation. The results highlight that embracing heterogeneity is not only necessary but also beneficial for achieving communication-efficient distributed computing.

Bibliography

[1] DeepSeek-AI, A. Liu, B. Feng, *et al.*, *Deepseek-v3 technical report*, 2025. [Online]. Available: `https://arxiv.org/abs/2412.19437`.

[2] D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan, "Pram programming: In theory and in practice," Concurrency: Practice and Experience, vol. 12, no. 4, pp. 211–230, Jun. 2000.

[3] L. G. Valiant, "A bridging model for parallel computation," Communications of the ACM, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[4] J. Fernández, J. Carretero, J. M. García, and F. Xhafa, "Measurement of the latency parameters of the multi-bsp model: A multicore benchmarking approach," The Journal of Supercomputing, vol. 67, pp. 565–584, Sep. 2014.

[5] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2007.

[6] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[7] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*. Jan. 1999.

[8] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," International Journal of High Performance Computing Applications, vol. 15, pp. 200–222, 2001.

[9] R. Buyya, "A taxonomy of computer-based grids," in Proc. IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGrid), IEEE, 2001, pp. 44–51.

[10] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[11] F. Gagliardi, L. Jones, B. Marten, A. Guarise, and R. Barbera, "The egee project: Status and perspectives," Journal of Grid Computing, vol. 3, no. 1–2, pp. 65–82, Jun. 2005.

[12] P. Mell and T. Grance, "The nist definition of cloud computing," Proc. National Institute of Standards and Technology(NIST), Tech. Rep., Sep. 2011.

[13] M. Armbrust, A. Fox, R. Griffith, *et al.*, "A view of cloud computing," Communications of the ACM, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[14] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Communications of the ACM, vol. 51, pp. 107–113, Jan. 2008.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in Proc. 2nd USENIX Conf. Hot Topics in Cloud Computing (HotCloud), Jun. 2010.

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Commun. ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Proc. USENIX Conf. Netw. Syst. Design Implement. (NSDI), Apr. 2012.

[18] D. Jiang, Y. Chen, and J. Chen, "Cloud log analysis with mapreduce," International Journal of Grid and Distributed Computing, vol. 3, pp. 17–26, Jun. 2010.

[19] C.-T. Chu, S. Kim, Y.-A. Lin, *et al.*, "Map-reduce for machine learning on multicore," in in Proc. Neural Information Processing Systems (NIPS), Dec. 2007, pp. 281–288.

[20] M. C. Schatz, "Cloudburst: Highly sensitive read mapping with mapreduce," Bioinformatics, vol. 25, pp. 1363–1369, Jun. 2009.

[21] Apache Software Foundation, *Apache hadoop 0.1.0 release notes*, `https://hadoop.apache.org/docs/r3.4.0/hadoop-project-dist/hadoop-common/release/0.1.0/CHANGELOG.0.1.0.html`, Apr. 2006.

[22] R. Gu, Y. Tang, C. Tian, *et al.*, "Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms," IEEE Trans. Parallel Distrib. Syst., vol. 28, no. 9, pp. 2539–2552, 2017.

[23] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," IEEE Trans. on Inform. Theory, vol. 60, pp. 2856–2867, May 2014.

[24] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," IEEE Trans. on Inform. Theory, vol. 64, pp. 109–128, Jan. 2018.

[25] O. O'Malley, "Terabyte sort on apache hadoop," Yahoo! Tech. Rep., May 2008.

[26] S. Li, S. Supittayapornpong, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded terasort," in Proc. 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2017.

[27] J. Xu, Q. Yu, X. Zhu, Q. Liu, and M. Chen, "Heterogeneous coded computing with heterogeneous communication costs," IEEE Trans. Parallel Distrib. Syst., vol. 32, pp. 1683–1697, Jul. 2021.

[28] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, “Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads,” in Proceedings of the VLDB Endowment, vol. 5, 2012, pp. 1802–1813.

[29] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Effective straggler mitigation: Attack of the clones,” in Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI), Apr. 2013.

[30] M. J. Flynn, “Some computer organizations and their effectiveness,” IEEE Transactions on Computers, vol. C-21, pp. 948–960, Sep. 1972.

[31] G. D. Ghare and S. T. Leutenegger, “Improving speedup and response times by replicating parallel programs on a snow,” in Proc. Job Scheduling Strategies for Parallel Processing (JSSPP), Jun. 2004.

[32] G. Ananthanarayanan, S. Kandula, A. Greenberg, *et al.*, “Reining in the outliers in map-reduce clusters using mantri,” in Proc. the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Oct. 2010.

[33] D. Wang, G. Joshi, and G. Wornell, “Using straggler replication to reduce latency in large-scale parallel computing,” SIGMETRICS Perform. Eval. Rev., vol. 43, pp. 7–11, Nov. 2015.

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

[35] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” IEEE Trans. on Inform. Theory, vol. 64, pp. 1514–1529, Mar. 2018.

[36] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, “A unified coding framework for distributed computing with straggling servers,” in Proc. IEEE Globecom Workshops (GC Wkshps), IEEE, Dec. 2016.

[37] M. Blaum and R. M. Roth, "On lowest density mds codes," IEEE Transactions on Information Theory, vol. 45, pp. 46–59, Jan. 1999.

[38] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," in Proc. IEEE Int. Symp. on Information Theory (ISIT), Jul. 2017.

[39] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: An optimal design for high-dimensional coded matrix multiplication," in Proc. Advances in Neural Information Processing Systems (NIPS), Dec. 2017.

[40] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding," IEEE Trans. on Inform. Theory, vol. 66, pp. 1920–1933, Mar. 2020.

[41] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," in Proc. Annu. Allerton Conf. on Communication, Control, and Computing, Oct. 2017.

[42] S. Wang, J. Liu, and N. Shroff, "Coded sparse matrix multiplication," in Proc. Int. Conf. on Machine Learning (ICML), Jul. 2018.

[43] S. Dutta, V. R. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in Proc. Advances in Neural Information Processing Systems (NeurIPS), Dec. 2016.

[44] S. Wang, J. Liu, N. Shroff, and P. Yang, "Fundamental limits of coded linear transform," Apr. 2018.

[45] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," in Proc. IEEE Int. Symp. on Information Theory (ISIT), Jun. 2017.

[46] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded fourier transform," in Proc. Annu. Allerton Conf. on Communication, Control, and Computing, Oct. 2017.

[47] A. Mallick and R. Pedarsani, “Rateless coded matrix multiplication for distributed computing,” IEEE J. Sel. Areas Inf. Theory, vol. 2, pp. 719–729, Sep. 2021.

[48] A. Severinson, A. Graell i Amat, and E. Rosnes, “Block-diagonal and lt codes for distributed computing with straggling servers,” Dec. 2017.

[49] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, “Gradient coding: Avoiding stragglers in distributed learning,” in Proc. Int. Conf. on Machine Learning (ICML), Aug. 2017.

[50] N. Raviv, I. Tamo, R. Tandon, and A. G. Dimakis, “Gradient coding from cyclic mds codes and expander graphs,” IEEE Trans. on Inform. Theory, vol. 66, pp. 7475–7489, Dec. 2020.

[51] W. Halbawi, N. Azizan, F. Salehi, and B. Hassibi, “Improving distributed gradient descent using reed-solomon codes,” in Proc. IEEE Int. Symp. on Information Theory (ISIT), Jun. 2018.

[52] S. Li, S. M. M. Kalan, A. S. Avestimehr, and M. Soltanolkotabi, “Near-optimal straggler mitigation for distributed gradient methods,” in Proc. IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW), May 2018.

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

[54] Q. Yan, S. Yang, M. Wigger, and X. Tang, “A group-based coded mapreduce scheme for reducing communication load,” in Proc. IEEE International Symposium on Information Theory (ISIT), IEEE, Jun. 2015.

[55] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, “A coded distributed computing scheme with resolvable design,” in Proc. IEEE International Symposium on Information Theory (ISIT), IEEE, Jun. 2017.

[56] M. A. Attia and R. Tandon, "Information theoretic limits of data shuffling for distributed learning," in Proc. IEEE Global Communications Conference (GLOBECOM), Dec. 2016.

[57] S. Gupta and V. Lalitha, "Locality-aware hybrid coded mapreduce for server-rack architecture," in Proc. IEEE Information Theory Workshop (ITW), IEEE, Nov. 2017.

[58] J. Jiang and L. Qu, "Cascaded coded distributed computing schemes based on placement delivery arrays," IEEE Access, vol. 9, pp. 152 623–152 633, Nov. 2021.

[59] Q. Yan, X. Tang, and Q. Chen, "Placement delivery array and its applications," in Proc. IEEE Information Theory Workshop (ITW), IEEE, Nov. 2018.

[60] V. Ramkumar and P. V. Kumar, "Coded mapreduce schemes based on placement delivery array," in Proc. IEEE International Symposium on Information Theory (ISIT), IEEE, Jul. 2019.

[61] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Compressed coded distributed computing," in Proc. IEEE International Symposium on Information Theory (ISIT), Jun. 2018.

[62] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in Proc. Advances in Neural Information Processing Systems 30 (NIPS), 2017.

[63] L. Song, C. Fragouli, and T. Zhao, "A pliable index coding approach to data shuffling," IEEE Transactions on Information Theory, vol. 66, pp. 1333–1353, Mar. 2020.

[64] F. Haddadpour, Y. Yang, V. Cadambe, and P. Grover, "Cross-iteration coded computing," in Proc. 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton), Oct. 2018.

[65] S. Prakash, A. Reisizadeh, R. Pedarsani, and S. Avestimehr, "Coded computing for distributed graph analytics," in Proc. IEEE International Symposium on Information Theory (ISIT), Jun. 2018.

[66] S. R. Srinivasavaradhan, L. Song, and C. Fragouli, "Distributed computing trade-offs with random connectivity," in Proc. IEEE International Symposium on Information Theory (ISIT), Jun. 2018.

[67] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded distributed computing: Straggling servers and multistage dataflows," in Proc. 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton), Sep. 2016.

[68] N. Woolsey, R. Chen, and M. Ji, "Cascaded coded distributed computing on heterogeneous networks," in Proc. 2019 IEEE Int. Symp. Inf. Theory (ISIT), 2019, pp. 2644–2648.

[69] K. Wan, M. Ji, and G. Caire, *Topological coded distributed computing*, arXiv preprint arXiv:2004.04421, `https://arxiv.org/abs/2004.04421`, 2020.

[70] W. Xia, P. Zhao, Y. Wen, and H. Xie, "A survey on data center networking (dcn): Infrastructure and operations," IEEE Communications Surveys & Tutorials, vol. 19, no. 1, pp. 640–656, Jan. 2017.

[71] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in Proc. ACM SIGCOMM Conf. Data Commun., Aug. 2008, pp. 63–74.

[72] J. Chung, K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Ubershuffle: Communication-efficient data shuffling for sgd via coding theory," in Proc. Adv. Neural Inf. Process. Syst. (NIPS), Dec. 2017.

[73] K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Coded computation for multicore setups," in Proc. IEEE Int. Symp. Inf. Theory (ISIT), Jun. 2017.

[74] A. Reisizadeh, A. Mokhtari, H. Hassani, A. S. Avestimehr, and R. Pedarsani, "Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization," in Proc. Int. Conf. on Learning Representations (ICLR), Apr. 2020.

[75] E. C. Ozfatura and D. Gündüz, "Coded federated computing in wireless networks with stragglers," IEEE J. Sel. Areas Commun., vol. 38, pp. 2526–2537, Oct. 2020.

[76] S. Li and S. Avestimehr, "Hierarchical coded computing," IEEE Trans. on Inform. Theory, vol. 67, pp. 7918–7935, Dec. 2021.

[77] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," IEEE Internet of Things Journal, vol. 3, pp. 637–646, Oct. 2016.

[78] M. Fahmideh, F. Haddadpour, M. B. Khuzani, and R. Pedarsani, "Serverless computing meets coded computing: A scalable and communication-efficient paradigm," IEEE/ACM Trans. Netw., vol. 30, pp. 2533–2546, Dec. 2022.

[79] S. Wang, A. Reisizadeh, A. Mokhtari, R. Pedarsani, and S. Avestimehr, "Coding techniques for communication reduction in serverless computing," in Proc. the IEEE International Symposium on Information Theory (ISIT), 2021.

[80] X. Wang, K. Lee, and S. Avestimehr, "Coding techniques for serverless computing: A survey," IEEE Commun. Surv. Tutor., vol. 23, pp. 2244–2274, Oct. 2021.

[81] T. Harter, D. Borthakur, S. Dong, *et al.*, "Analysis of HDFS under HBase: A facebook messages case study," in Proc. 12th USENIX Conference on File and Storage Technologies (FAST '14), Feb. 2014.

[82] L. Tang and A. Ramamoorthy, "Low subpacketization schemes for coded caching," in Proc. 2017 IEEE International Symposium on Information Theory (ISIT), Sep. 2017, pp. 2790–2794.

[83] K. Shanmugam, A. M. Tulino, and A. G. Dimakis, "Coded caching with linear subpacketization is possible using ruzsa-szemerédi graphs," in Proc. 2017 IEEE International Symposium on Information Theory (ISIT), Jan. 2017.

[84] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in Proc. the 2012 ACM SIGMOD International Conference on Management of Data, May 2012.

[85] B. Pustina, "Review of performance evaluation benchmarks for apache hadoop," in Politecnico di Milano Thesis, Jun. 2014.

[86] O. O'Malley, "Winning a 60 second dash with a yellow elephant," Yahoo! Tech. Rep., 2009, Technical report.

[87] *Sort benchmark*, `http://sortbenchmark.org/`.

[88] Y. Tao, W. Lin, and X. Xiao, "Minimal mapreduce algorithms," in Proc. the 2013 ACM SIGMOD International Conference on Management of Data, ACM, Jun. 2013.

[89] D. Lemire, "Sorting 1 terabyte in 209 seconds," in *Blog Post / Technical Blog*, Jul. 2008.

[90] S. Bhardwaj, A. Singh, and Vanraj, "Analyzing bigdata with hadoop cluster in hdinsight azure cloud," in Proc. 2015 Annual IEEE India Conference (INDICON), Jun. 2015.

[91] K. Slagter, C.-H. Hsu, Y.-C. Chung, and X. Zhang, "An improved partitioning mechanism for optimizing massive data analysis using mapreduce," in Journal of Supercomputing, Apirl 2013.

[92] F. M. Matu, "Optimized TeraSort algorithm for data analytics: Case of climate data analysis," Thesis, Strathmore University, Faculty of Information Technology (FIT), Nairobi, Kenya, Jun. 2017.

[93] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," ACM SIGCOMM Comput. Commun. Rev., vol. 41, pp. 98–109, Apr. 2011.

[94] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Proc. Advances in Neural Information Processing Systems (NIPS), Dec. 2012.

[95] N. Karamchandani, U. Niesen, M. A. Maddah-Ali, and S. N. Diggavi, "Hierarchical coded caching," IEEE Transactions on Information Theory, vol. 62, pp. 3212–3229, Jun. 2016.

[96] C. A. Floudas, *Nonlinear and Mixed-Integer Optimization: Fundamentals and Applications*. Oxford University Press, 1995.

[97] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. Wiley, 1988.

[98] S. Weise, "The mixed-integer conic optimizer in mosek," in Proc. 21st Int. Symp. Math. Program., Aug. 2021, pp. 1–2.

[99] J. E. Mitchell, "Branch-and-cut algorithms for combinatorial optimization problems," in *Handbook of Applied Optimization*, Oxford University Press, Apr. 2000.

[100] F. Xu, S. Shao, and M. Tao, "New results on the computation-communication tradeoff for heterogeneous coded distributed computing," IEEE Transactions on Communications, vol. 69, pp. 2254–2270, Apr. 2021.

[101] D. S. Johnson, "Fast algorithms for bin packing," Journal of Computer and System Sciences, vol. 8, pp. 272–314, Jun. 1974.