On Resource Allocation in Time-constrained Coded Distributed Computing Systems

by

Mehrad Mehrabi

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Communications

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Distributed computing systems have been widely used in recent years to handle massive computations required by newly emerged machine learning algorithms and signal processing problems. Also, the use of error correction codes has been proposed to mitigate the negative impact of slow workers by adding redundancy to the computational tasks. In practice, a distributed computing system often receives multiple tasks each needs to be finished by a specific deadline. Furthermore, service providers may offer different levels of service based on their users' subscription tiers. In this thesis, we first consider a scenario where multiple matrix-vector multiplication jobs arrive in a distributed computing system. The main challenges in such a system are random task arrivals and random execution times due to the slow workers. To address these challenges, we present two algorithms to maximize the number of tasks completed before their deadlines. Then, we study a tiered time-constrained distributed computing system, where there are multiple users with distinct subscription classes and each with a time-sensitive computational job. We assume the system receives rewards for finishing tasks prior to their deadlines and gives higher priority to tasks associated with higher subscription tiers. To maximize the overall reward of the system, we present a worker assignment scheme. In both studies, it is shown that our proposed algorithms provide comparable performances to unachievable upper bounds while exhibiting significantly lower complexity.

Preface

The results and algorithms of Chapter 3 has been published in the IEEE Signal Processing Letters under the title "Deadline-Aware Coded Computation Across Homogeneous Workers". Also, the results of Chapter 4 were presented in the 31st Biennial Symposium on Communications 2023 (IEEE BSC 2023).

"To my family for their love and heartwarming support"

Acknowledgements

First and foremost, I am deeply grateful to my supervisor, Dr. Masoud Ardakani, whose guidance, expertise, and invaluable feedback have been instrumental in shaping the direction of this research. I have been truly inspired by his unwavering support throughout this past three years. Second,I want to thank the committee members, Dr. Xingyu Li and Dr. Yindi Jing for their constructive ideas and dedicating their time, reading my thesis.

Also, I would like to thank my family for all the support they have given me throughout my life. Lastly, I wish to extend my appreciation to my friends in Edmonton, whose presence made my time in the city truly enjoyable.

Table of Contents

List of Tables

List of Figures

Abbreviations

CDC Coded Distributed Computing.

CPU Central Processing Unit.

FEC Forward Error Correction.

IoT Internet of Things.

MDS Maximum Distance Separable.

SJF Shortest Job First.

Chapter 1

Introduction

1.1 Motivation

The demand for massive computation has been steadily increasing in recent years. As technology advances and new applications emerge, there is a growing need for processing vast amounts of data and performing complex calculations at an unprecedented scale. Signal processing, machine learning [1], and data analytic problems heavily rely on massive computation to extract valuable insights. Additionally, the proliferation of connected devices, the Internet of Things (IoT) [2], and the advent of 5G networks [3] have led to an exponential growth in data generation, further fueling the demand for powerful computing resources.

These applications involve processing large amounts of data and performing complex calculations, which can overwhelm the capabilities of a single processing unit.

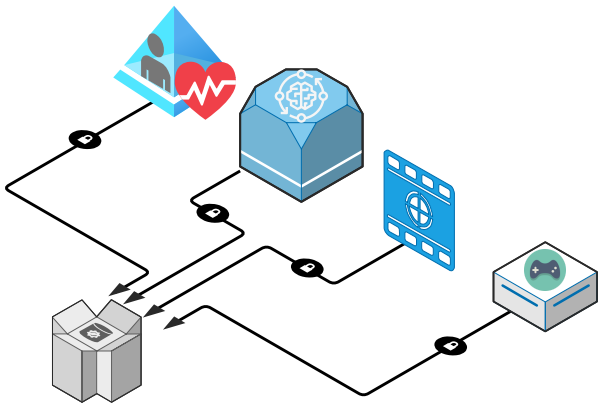

Figure 1.1: The growing demand for massive computations

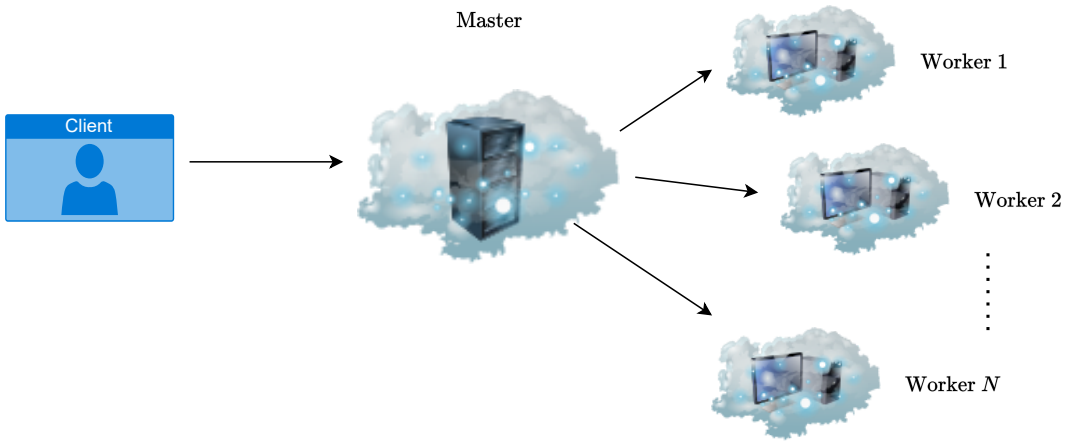

Figure 1.2: An schematic of a general distributed computing system

Additionally, the time taken to complete these computations would be impractical if performed sequentially on a single processing unit. Accordingly, a time-sensitive computational task may not meet its deadline if it is performed in such a limited unit [4]. Also, a single processing unit has a restricted memory capacity and limited processing power. Large-scale computations often require significant amounts of memory to store and manipulate data efficiently [5, 6]. Hence, Single processing units may not have the necessary resources to handle such demands. This has led to the wide use of distributed computing for large-scale computations in recent years [7–9].

Distributed computing systems are a paradigm of computing that involve multiple interconnected computers or processors working together to solve complex problems or perform large-scale computations [10]. Instead of relying on a single central processing unit (CPU), distributed computing systems distribute the workload across multiple nodes, which can be individual machines, servers, or even clusters of computers. By distributing the computation across these nodes, tasks can be executed concurrently, reducing the processing time and enabling the system to handle massive computational tasks effectively [11].

As depicted in Fig. 1.2, generally in distributed computing systems, a centralized controller, namely the master, is responsible for handling the computational tasks

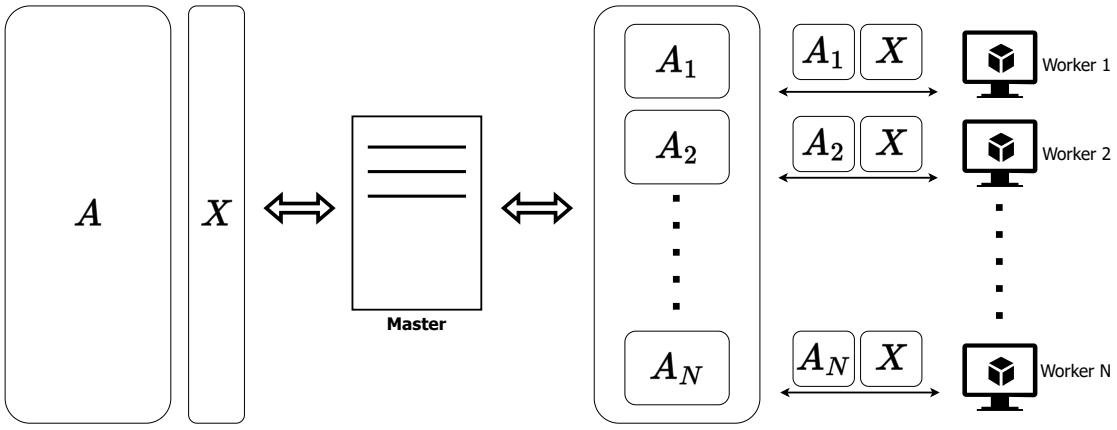

Figure 1.3: Matrix-vector multiplication in a distributed computing system with N workers

arrive at the system. In particular, the master divides the task into several sub-tasks. Afterward, the sub-tasks that are generated are allocated to the available processing units, known as worker nodes, within the system. By distributing the workload among multiple workers in parallel, the overall execution time is significantly reduced.

An example of a computational task is illustrated in Fig. 1.3. The matrix-vector multiplication $A_m x_m$ arrives at the system. Then the master divides the matrix A_m into N sub-matrices. The generated sub-matrices, then are distributed among the N workers available in the system. The master is able to recover the final result once it receives the outcome of all sub-tasks from the workers.

Distributed computing systems offer several advantages. The use of computing nodes with low-cost hardware and the possibility to easily add them, respectively, make these systems economical and scalable[12]. Thus, distributed computing is adopted in the computing services and real-life applications such as wireless sensor networks [13], online games [14], video streaming [15], distributed database management systems [16], and real-time process control [17].

Although, many advantages are offered by distributing computing, there are challenges such as time-sensitivity [18], slow workers [19] and heterogeneity of the computing nodes that need to be addressed in practice. In particular, large number of

computing nodes in a distributed computing system may have different storage capacity, computing power, and network resources. This emphasizes the importance of proper resource allocation schemes in order to improve the performance of these systems.

1.2 Related Work

Distributed computing systems strive to enhance computation speed by distributing the workload across multiple worker nodes, resulting in accelerated computations. The existence of straggling nodes [20] which experience unpredictable performance failure or slowdown, however, may prolong computation time [21]. The straggling problem can be attributed to various factors, including the sharing of resources, hardware failures, and uneven distribution of workloads. [22].

To mitigate the negative impact of stragglers, some form of computation redundancy is added. The traditional approach involves naive replication of tasks to mitigate the stragglers [23]. However, recent results regarded the coding theoretic technique as a promising solution to more effectively address the challenges in distributed computing [24], [25]. The combination of coding and distributed computing is termed coded distributed computing (CDC).

Besides mitigating stragglers' effects, two important aspects to improve the performance of distributed computing systems are task assignment and resource allocation which have been the subject of many recent studies [26–32]. Load allocation is one of the main challenges in distributed computing systems. Specifically, proper load distribution among the workers can significantly affect the overall computation time. Therefore, much research efforts have been dedicated to this matter.

Furthermore, it is reasonable to assume that computational tasks in commercial distributed computing systems have time-sensitive characteristics. These tasks require timely execution and completion to ensure the system meets its objectives effectively. Failure to meet the time requirements of these tasks can result in degraded

system performance and missed deadlines. To meet the time constraints, distributed computing systems employ various techniques such as task scheduling and resource allocation. For example [33–35] study deadline constrained task scheduling in cloud computing systems.

1.3 Thesis Overview

In this thesis, we study two problems. First, we consider a task scheduling problem in a homogeneous distributed computing system. In particular, we assume multiple time-sensitive computational tasks arrive at the system at random time instances. For this setup, we propose two task scheduling algorithms to maximize the number of tasks completed before their deadlines. Furthermore, we apply a proper coding scheme to alleviate the impact of stragglers.

Second, we focus on a tiered heterogeneous CDC system in which we design a deadline-aware worker assignment scheme in order to maximize the system's reward. In particular, we consider a system that consists of multiple users and several workers, each with different processing capabilities. The system is rewarded by users, for being on-time, i.e. completing the task before its deadline, and also based on the priority of the user's task. For this system, we propose a worker assignment scheme to maximize the total reward achieved by the system.

The key novelty of this thesis can be summarized as studying multi-user deadline constrained distributed computing systems. Our primary contributions involve introducing task scheduling and worker assignment policies with low complexity in these systems. In designing these policies, however, several difficulties arise. These challenges are discussed throughout this work.

The remainder of this thesis is organized as follows. In Chapter 2, we provide the required background on coding theory which forms the basis for understanding the concept of coded distributed computing. Also, we conduct a comprehensive review of the relevant literature pertaining to resource allocation in coded distributed comput-

ing systems. In Chapter 3, we study the problem of task scheduling in a homogeneous distributed computing system, where each task has its execution deadline. In Chapter 4 we focus on the problem of worker assignment in a heterogeneous system, and propose a worker assignment scheme to maximize the total reward of the system. Finally, in Chapter 5, we conclude this thesis and provide some possible future research directions.

Chapter 2

Background

2.1 Forward Error Correction Codes

Forward Error Correction (FEC) codes are an integral part of many communication systems and data storage technologies. These codes are specifically designed to detect and correct errors that may occur during data transmission or storage. By introducing redundancy into the transmitted data, FEC codes enable the receiver to reconstruct the original information even if some bits are corrupted or lost in the process.

To provide an illustration, let us consider an information block consists of k bits. In the encoding process, $n\,(n > k)$ symbols are generated from original k bits. This process is known as encoding with an (n, k) code, where n represents the size and k represents the dimension of the code. In the given code, the coding rate is defined as $r \triangleq \frac{k}{n}$. Consequently, through the decoding process, the receiver can recover the original k bits from the data block that may have been corrupted by errors. In coding theory, we work with elements that are members of a finite field F.

Definition 1 (Finite Field $\mathbb{F}_q$) *A finite field $\mathbb{F}_q$, also referred to as a Galois field $GF(q)$, is a finite set of q elements if $(\mathbb{F}, +, \times)$ has the following properties:*

1. *F and "$+$" form an Abelian group with the additive identity element "0".*
2. *F and "$\times$" form an Abelian group with the multiplicative identity element "1".*
3. $(a + b) \times c = a \times c + b \times c$, $\forall a, b, c \in \mathbb{F}$

In a finite field F, a linear (n, k) code generates $n > k$ symbols, through linear operations on the original k symbols. These linear codes can be represented by a generator matrix $G^{n\times k}$, and the encoding process is described by the equation $y = Gx$, where $x^{k\times 1}$ represents the original data vector and $y^{n\times 1}$ represents the resulting coded block. Each coded block is referred to as a codeword, and there exist 2^k different codewords in total.

2.1.1 MDS codes

A significant characteristic of linear codes is their minimum distance, which represents the minimum Hamming distance between any two codewords.

Definition 2 (Hamming weight and Hamming distance) *Hamming weight refers to the number of non-zero bits in a binary sequence. Assuming* $\mathbf{a}$ *is a binary vector,* $wt(\mathbf{a})$ *denotes Hamming weight of* $\mathbf{a}$*. Accordingly, Hamming distance between two vectors* $\mathbf{a}$ *and* $\mathbf{b}$*,* $wt(\mathbf{a}, \mathbf{b})$*, is denoted as* $d(\mathbf{a}, \mathbf{b})$*.*

A code with minimum distance d tolerates any $d - 1$ symbol erasures. Hence, in case that $d - 1$ symbols of the coded block is missing, the original block can still be recoverd.

Theorem 1 *In an* (n, k) *code, the minimum distance* d *is bounded to*

$$d \leq n - k + 1 \tag{2.1}$$

This bound is known as the Singleton bound [36]. The codes that achieve Singleton bound with equality are called the maximum distance separable (MDS) codes. Since the existence of at least $n - (d - 1)$ symbols is necessary for recovering the original block, $k + n - (d - 1)$ redundant symbols are needed for decoding. As MDS codes achieve the equality in (1), they can tolerate up to $n - k$ erasures. This implies that the decoding process will be successful with any combination of k coded symbols. Hence, the decoding overhead is zero. This has resulted in MDS codes being widely used as popular coding schemes in distributed computing systems [22, 31, 37, 38].

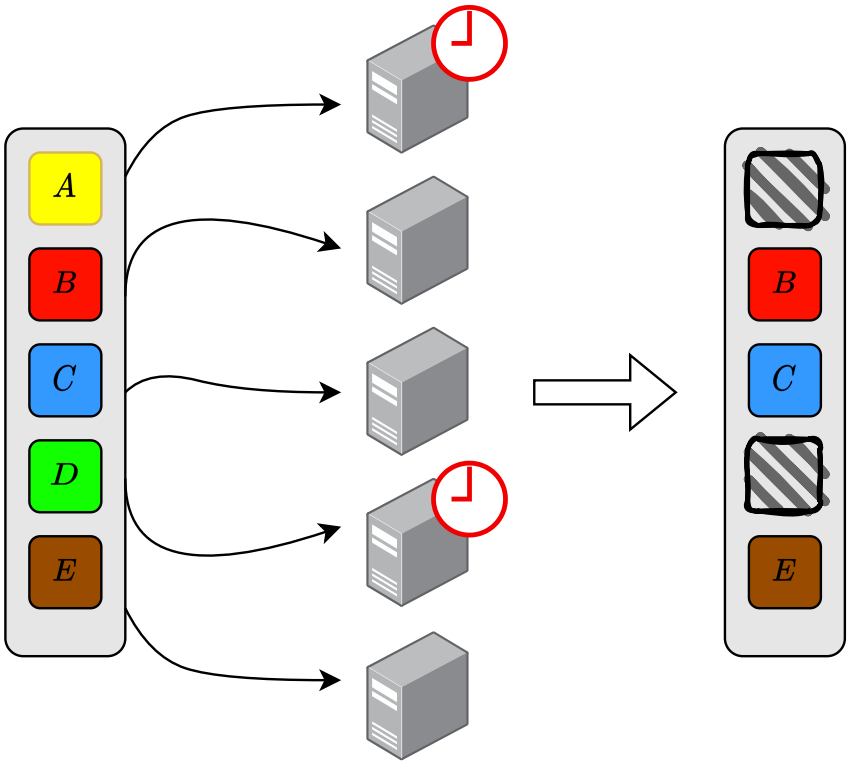

Figure 2.1: Distributed computation with stragglers

2.2 Coded Distributed Computing (CDC)

As mentioned previously, one of the primary obstacles in distributed computing systems is the presence of stragglers. Even though distributed computing systems aim to speed up computations, slow workers cause a bottleneck that prevents faster computations. Taking Fig. 2.1 as an illustration, suppose the computation completes once all five sub-tasks are finished. Nevertheless, workers responsible for computing sub-tasks A and D encounter an unusual delay in comparison to the other three workers. As a result, the computation is prolonged because we must wait for the results from those two workers to finalize the overall computation.

While trivial techniques such as work exchange [39] and repetition [40] have been suggested to address the stragglers, these approaches involves inefficient redundancy or coordination among nodes, which significantly increases communication costs and computation loads [41]. This highlights the necessity for a novel technique that can

more efficiently and comprehensively tackle the challenges posed by stragglers.

Similar to the application of coding theoretic techniques in communication systems to address channel impairments, a similar approach can be employed in distributed computing systems to alleviate the impact of stragglers. In essence, coding theoretic approaches are employed to incorporate redundant information into transmitted messages, enabling the receiver to utilize it for error correction purposes. Likewise, coding solutions have recently been suggested to overcome the challenges caused by slow workers in distributed computing systems. Coded distributed computing (CDC) is the term used as the combination of coding techniques and distributed computing.

CDC is a promising solution to mitigate the challenges in distributed computing systems. Generally, in coded distributed computing systems a task is divided into $k < n$ sub-tasks, where n is the number of computing nodes, and introduces redundancy by using error correction codes such that by completion of any fixed-cardinality subset of the sub-tasks, referred as recovery threshold, the desired solution can be realized. Referring back to the previous example in Fig. 2.1, let us assume the original task is now divided into three sub-tasks instead of five. Then, using a $(5, 3)$ code, five coded sub-tasks are generated and sent to the workers. Consequently, the master is now able to recover the final result upon collecting the results of three fastest workers. Since it is not required to wait for the outputs of all computing nodes, CDC significantly reduces the computation time. Therefore, different encoding schemes for various conditions and computational jobs are proposed [22, 37, 42–55].

Matrix-vector multiplication is an illustrative example of a frequently performed task in distributed computing systems. Matrix-vector multiplication is a fundamental operation that forms the backbone of numerous data analytics applications such as machine learning and signal processing. These applications often involve the processing of enormous volumes of data, necessitating substantial computational and storage resources that exceed the capabilities of a single processing unit. Consequently, several algorithms have been developed to speed-up matrix-vector multiplication by

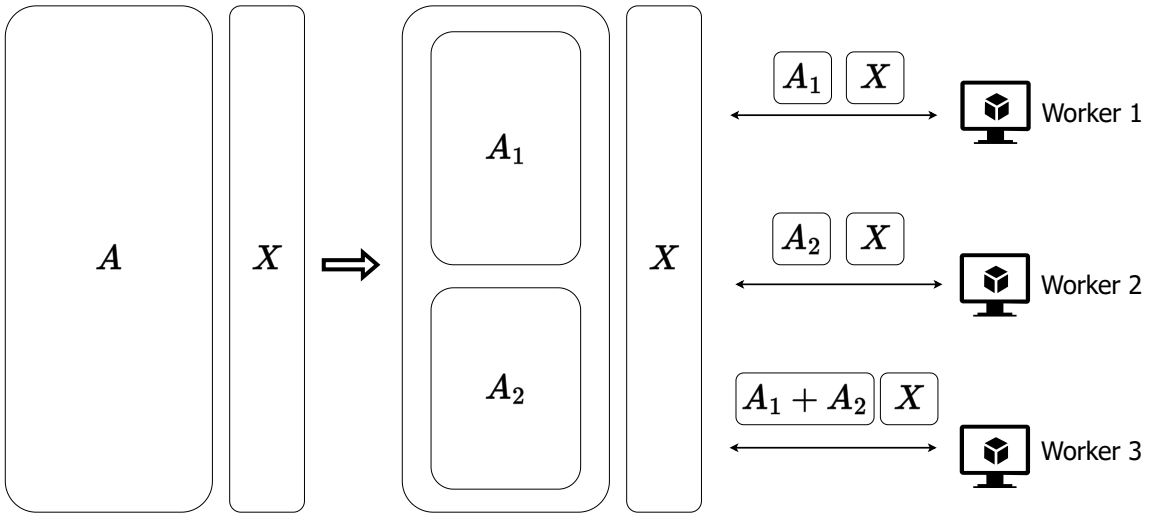

Figure 2.2: Coded distributed matrix-vector multiplication over 3 workers

distributing the computational workload among multiple computing nodes [53, 56, 57].

Moreover, MDS codes have recently been utilized in distributed systems to accelerate the computation of matrix-vector products [22, 42, 48]. An example of a matrix-vector multiplication with MDS coding is illustrated in Fig. 2.2, where a $(3, 2)$ MDS code is employed to compute the multiplication of matrix A with vector x, in a system with three workers [53]. First, the matrix A is horizontally divided into two sub-matrices such that $A = [A_1^T A_2^T]^T$. Then the vector x and sub-matrices A_1, A_2 and $A_1 + A_2$ are sent to the workers, respectively. Ax can be obtained upon collecting the results of any two workers.

2.3 Resource Allocation

In distributed computing systems, optimizing the computational loads allocated to workers is essential for improving overall system performance and efficiency. It is important to minimize the average execution time for tasks distributed among multiple workers. To achieve this, load allocation algorithms have been proposed, aiming to balance the workload among workers effectively.

One of the main challenges in load allocation arises when workers have different computational capabilities. Some workers may be more powerful and efficient than others, capable of completing assigned sub-tasks faster. On the other hand, weaker workers might take longer to finish the same computational job. As a result, a proper load allocation becomes even more crucial in such heterogeneous environments.Various load allocation algorithms have been proposed to address this issue. These algorithms take into account factors such as worker capabilities and computation workloads. The objective is to intelligently distribute tasks to workers in a way that minimizes the average task execution time.

In [27], the authors find an optimal load allocation policy for a matrix-vector multiplication under a heterogeneous scenario where multiple worker nodes with varying computational properties exist in the system. Similarly, optimal load allocation for heterogeneous clusters is proposed in [30], where workers with similar computational capacities form a cluster and several clusters exist. Hence, workers from different clusters may have completely different parameters. In [27, 30] it is shown that optimal load allocation can significantly reduce the execution time in distributed computing systems.

Although [27] and [30] work on the load allocation problem for a single computational task, in practice workers may be assigned to multiple tasks. In that case, how tasks arrive at the system affects the load allocation algorithm. One-shot task arrival, where multiple tasks arrive at the master simultaneously, is considered in [31]. The authors considered a joint worker assignment and load allocation problem in a heterogeneous network of workers. The objective in [31] is to minimize the communication plus computation delay of tasks.

In contrast, [29] examines an alternative model for task arrival, where tasks arrive at random times. Authors have considered an online task assignment problem in a heterogeneous coded distributed computing system. To tackle the difficulty of finding the optimal solution, they proposed an approximate online algorithm that relies on

convex optimization and time recursion.

Also, artificial intelligence solutions have been studied in the literature. For example, [32] proposes a reinforcement learning based resource allocation scheme for a hybrid environment, i.e. deals with a combination of time-critical and regular non-time-critical applications. It has been shown that the proposed model can reduce both missing deadline occurrences for time-critical applications and the average completion delay for all jobs.

2.4 Task scheduling

Task scheduling is the process of allocating and managing resources to execute a set of tasks in computing system. It involves determining the order in which tasks should be executed and allocating appropriate resources to each task.

In a time-constrained environment, task scheduling is carried out with the objective of maximizing the number of tasks that are finished before their deadlines. Thus, through task scheduling, we are making decisions about which tasks should run, and when they should run. Typically, the task scheduler maintains a queue of tasks awaiting execution, known as the task queue. The scheduler selects tasks from this queue and determines which task should be given access to system resources next.

An example of a task scheduling problem is illustrated in Fig. 2.3. We assume the previous job is finished at T_{current}, the current time of the system. In the meantime, three jobs j_1, j_2 and j_3 have arrived at the system at $T_{\text{current}} - 3$, $T_{\text{current}} - 2$ and $T_{\text{current}} - 1$, respectively. Their deadlines are $T_{\text{current}} + 5, T_{\text{current}} + 6$ and $T_{\text{current}} + 8$. Assuming execution time is deterministic, it will take 5, 2, and 3 units of time to finish j_1, j_2, and j_3, respectively. Based on the deadlines and computation times, it is obvious that the system is able to finish at most two tasks before their deadlines. To do so, two options are available: (1) compute j_1, then j_3. (2) compute j_2, then j_3. As such, a task scheduling algorithm may choose option (1) or (2) as its final schedule.

Task scheduling in distributed computing refers to the process of allocating and

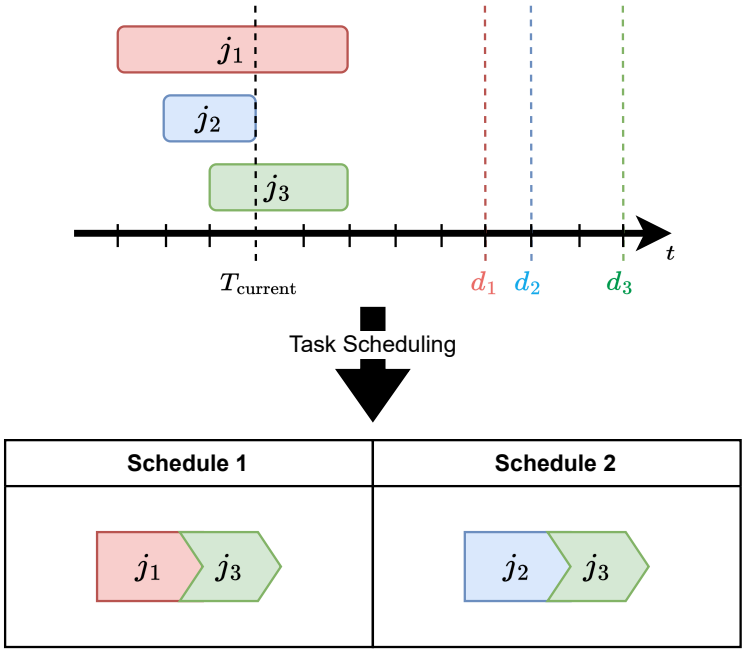

Figure 2.3: A toy model of application of task scheduling

managing tasks across multiple worker nodes in a distributed system. The task scheduler in a distributed system is responsible for determining which tasks should be executed on which nodes, taking into account factors such as the computational capabilities of the workers and the time constraints associated with the tasks.

2.4.1 Shortest Job First (SJF)

One of the most commonly used task scheduling algorithms is Shortest Job First (SFJ). The SJF algorithm, is a task scheduling algorithm that prioritizes tasks based on their expected execution time, with the objective of minimizing the average waiting time for each task until its execution is finished. In SJF, the task with the shortest estimated duration is given the highest priority and scheduled for execution first.

The SJF algorithm assumes that the duration of each task is known or can be accurately estimated in advance. It works by comparing the expected execution times of the tasks in the task queue and selecting the task with the shortest execution time to be executed next. If multiple tasks have the same expected execution time, the algorithm may use additional criteria, such as task arrival time or priority, to break

the tie.

In general, two types of SJF exist: non-preemptive and preemptive. In non-preemptive SJF scheduling, once a task starts executing, it is allowed to run until completion without interruption. Alternatively, in preemptive SJF, a running task may be interrupted or preempted if a new task arrives with a shorter expected execution time.

Let us consider a non-preemptive SJF scheduling in a deadline-aware system. The arrival time, execution time and deadline of each task is presented in Table 2.1.

Table 2.1: Task Queue

Task	Arrival Time	Execution Time	Deadline
A	2	3	6
B	1	4	7
C	6	3	9
D	0	4	5
E	8	2	11

At time = 0, task D arrives and starts execution. Tasks B and A arrive at time = 1 and time = 2, respectively, but D continues execution. At time = 4, D finishes its execution. As task A has a shorter execution time than B, it starts execution. It continues execution until time = 6. At time = 6, A reaches its deadline and has not yet been completed, so it would be terminated. Also, at this time, task C arrives and due to its shorter execution time than B, it starts execution. As a result, task B would miss its deadline. At time = 8, task E arrives, but C still needs 1 unit of time to complete. At time = 9, D is finished and E is executed. Finally, at time = 11, E finishes its execution. Thus, tasks D, C and E are completed before their deadlines.

Chapter 3

Deadline-Aware Coded Computation Across Homogeneous Workers

3.1 Introduction

As discussed, distributed computing systems have been the subject of many recent studies. Numerous studies have been conducted with the goal of enhancing the performance of distributed computing systems through the introduction of coding methods, load allocation policies, and worker assignment schemes. While these existing works (e.g., [29–31]) exclusively focus on tasks that are not time-sensitive, there are many practical applications where the computation is successful only if the result is sent to the user before a specific deadline [32–35].

In such scenarios, it is possible that the master may not be able to successfully complete all the tasks within their respective deadlines. Consequently, the implementation of task scheduling algorithms becomes crucial. These algorithms aim to plan the execution of tasks based on their workload and deadlines, maximizing the number of tasks that can be successfully completed.

A task scheduling algorithm takes into consideration both the load of tasks and their associated deadlines to make informed decisions. By prioritizing tasks with earlier deadlines or adjusting the order of task execution, the algorithm can optimize

the utilization of resources and increase the likelihood of meeting as many deadlines as possible.

Additionally, an efficient task scheduling algorithm can contribute to reducing wasteful operations. By identifying and avoiding tasks that are impossible to complete before their respective deadlines, the algorithm can prevent unnecessary computations and allocate resources more effectively. This optimization can significantly enhance overall system efficiency and improve the overall success rate of completing tasks within their prescribed time constraints.

In this chapter, we study task scheduling in a homogeneous coded distributed system. In particular, we consider matrix-vector multiplication tasks with a random task arrival model and aim at maximizing the number of tasks completed before their deadlines. Our main challenge is the randomness in both task arrival and task execution time, where the latter is due to straggling. We present two sub-optimal greedy task scheduling algorithms, namely "simple greedy" and "farsighted greedy". To evaluate the performance of these two algorithms, we also consider a genie-aided algorithm that knows the exact arrival time as well as the exact execution time of all tasks. Using this knowledge, it finds the task scheduling with the maximum number of finished tasks. Hence, the performance of this genie-aided algorithm can be considered as an unachievable upper bound.

3.2 System Model

3.2.1 System architecture

We consider a homogeneous distributed computing system, where workers have similar computational capacities, with N workers, as shown in Fig. 3.1. Each task j_m, arrives at the system at time t_m, has a deadline d_m and is a matrix-vector multiplication $A_m x_m$, where $A_m \in \mathbb{R}^{l_m \times s_m}$ and $x_m \in \mathbb{R}^{s_m}$. Thus, the total time available for execution of task j_m is $d_m - t_m$. We denote the computational job j_m, with a tuple,

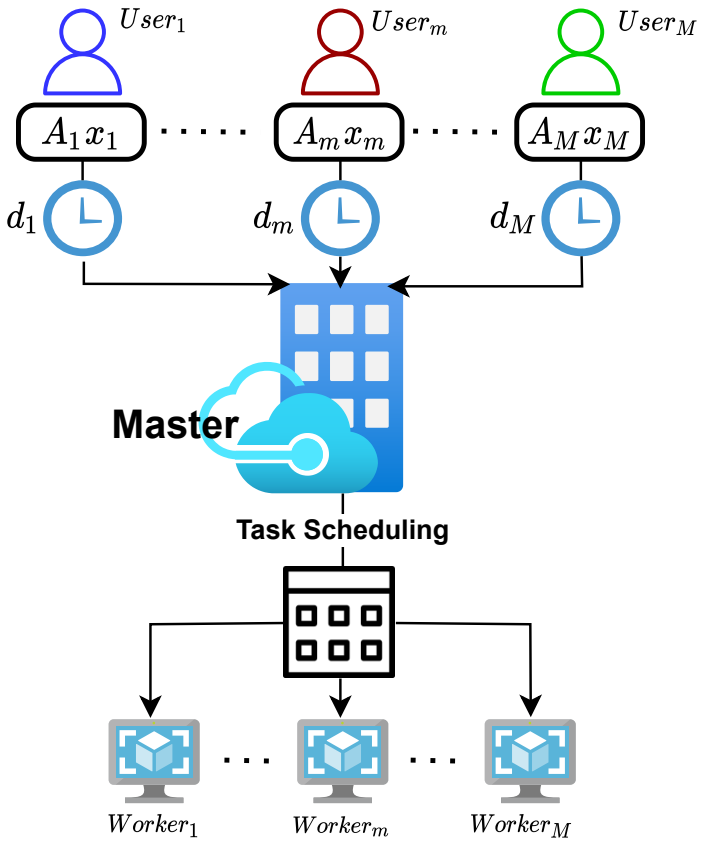

Figure 3.1: Illustration of a distributed computing system with multiple and time-critical tasks

$j_m = (t_m, d_m, l_m)$. Here, l_m—the number of inner products of this task—represents the job load. Furthermore, we define $J = \{(t_1, d_1, l_1), (t_2, d_2, l_2), ..., (t_M, d_M, l_M)\}$, as the set of jobs that have already arrived at the system and are awaiting scheduling and execution. Please note that the arrival of a new task results in updating J.

To mitigate the straggler effect, we use maximum-distance separable (MDS) codes. To this end, matrix A_m is first horizontally divided into K sub-matrices, $A_{m,k} \in \mathbb{R}^{\frac{L_m}{K} \times S_m}$, $k \in \{1, 2, \cdots, K\}$. Then, these K sub-matrices are encoded by an (N, K) MDS code, where N is the number of workers. Worker n, $n \in \{1, 2, \cdots, N\}$, is then assigned $\tilde{A}_{m,n} \in \mathbb{R}^{\frac{L_m}{K} \times S_m}$ to compute its sub-task $\tilde{y}_n = \tilde{A}_{m,n} x_m$. Since the code is MDS, the master is able to recover the final result $y = A_m x_m$ upon collecting any K completed sub-tasks from the workers. We assume that encoding delay is negligible

compared to computation delay[27, 29, 30]. For instance, A_m may represent a training dataset which is encoded and available to workers ahead of computation.

3.2.2 Task execution model

We assume that the time that a worker needs to finish its assigned load is a random variable T_{comp} following a shifted exponential distribution [22, 27, 37, 55]. This model accurately captures the random behaviour of the computation time[22]. Thus, if we measure the computational load of a matrix-vector multiplication with the number of inner products to be performed, l, [29, 31], the probability that a worker finishes its allocated load before time t is given by

$$P(T_{comp} \leq t) = \begin{cases} 1 - e^{-\frac{\mu}{l}(t-\alpha l)}, & t > \alpha l, \\ 0, & otherwise, \end{cases} \tag{3.1}$$

where $\mu > 0$ is a parameter for modeling the straggler effect, and $\alpha > 0$ is the time required for computing one inner product. Since we consider a homogeneous environment, all workers have the same μ and the same α.

3.2.3 Task arrival model

In this chapter, we consider a random task arrival model. When tasks are initiated from independent users, it is reasonable to assume that their arrival time are totally random in time, meaning that an exponential distribution captures the time gap between consecutive task arrivals. Hence, here, we use an exponential distribution with parameter λ, where $\frac{1}{\lambda}$ is the average time gap between consecutive task arrivals. More specifically, assuming that the i-th task arrives at t_i, the next arrival will be at $t_{i+1} = t_i + \tau$, where τ is a random variable with the following distribution:

$$P(\tau \leq t) = \begin{cases} 1 - e^{-\lambda t}, & t \geq 0 \\ 0, & otherwise, \end{cases} \tag{3.2}$$

A similar task arrival model is used to model file queries from cloud storage systems [58, 59].

3.3 Problem Definition

Our objective is to develop task scheduling algorithms that maximize the number of tasks completed before reaching their deadline. Let us assume there is no randomness in the problem. That is, the exact arrival time of tasks are known in advance, and execution times are deterministic. Such deterministic scenarios are studied in the literature as they can properly model a master with a single user and dedicated servers. For example, [60, 61] considered a system in which n jobs each with a release time , deadline and processing time have to be scheduled on m identical machines. We call the algorithms presented in such setups "genie-aided". However, in a multi-user scenario, where tasks are requested at random times by different users and servers are subject to straggling (hence execution times are also random), the above assumptions are no longer valid and the genie-aided algorithm is inapplicable.

Therefore, in this chapter, we consider the randomness in both arrival time and execution time of the tasks and suggest two scheduling algorithms namely *simple greedy* and *farsighted greedy*. While our proposed algorithms are not aware of upcoming tasks, nor can they predict the exact run-time of tasks, they provide performances close to a genie-aided algorithm.

To handle random execution times in our proposed solutions, we use the average execution time to schedule tasks. In addition, since it is possible that during a task execution, new tasks arrive at the system, our algorithms should update the schedule after finishing each task.

Put it together, our proposed algorithms form a schedule based on the existing tasks and their average execution time. Next, the first scheduled task is run. Upon completion of the first task, the schedule is updated to make a new arrangement in the order of the tasks, considering the actual time consumed by the first task and the newly arrived tasks. This continues until all possible tasks are executed and there is no newly arrived task.

3.3.1 Average execution time

As briefly discussed above, for the purposes of scheduling, the average execution time is used in all of our proposed algorithms. In order to compute the average execution time, we use the task execution model introduced in Section 3.2.2. Since the computation time of a sub-task at a worker is independent from the others, using an (N, K) MDS code, the execution time of a task can be expressed as the K^{th} order statistic of individual worker run-times [22, 53].

Thus, the average execution time of the matrix-vector multiplication $A_m x_m$ is given by

$$\mathbb{E}[T_{\text{comp}}^{\text{MDS}} | A_m \in \mathbb{R}^{l_m \times s_m}] = \frac{l_m}{K} \left[\frac{(H_N - H_{N-K})}{\mu} + \alpha \right] \tag{3.3}$$

Where H_n is the Harmonic number and defined as $H_n \triangleq \sum_{i=1}^{n} \frac{1}{i}$. If n is large, we can approximate $H_n \simeq \log(n)$ [22]:

$$\mathbb{E}[T_{\text{comp}}^{\text{MDS}} | A_m \in \mathbb{R}^{l_m \times s_m}] = \frac{l_m}{K} \left[\frac{1}{\mu} \log \frac{N}{N-K} + \alpha \right] \tag{3.4}$$

Hence, Denoting the rate of the MDS code with $r \triangleq \frac{K}{N}$, Eq. (3.4) can be written as:

$$\mathbb{E}[T_{\text{comp}}^{\text{MDS}} | A_m \in \mathbb{R}^{l_m \times s_m}] = \frac{l_m}{Nr} \left[\alpha - \frac{1}{\mu} \log(1-r) \right] \tag{3.5}$$

3.4 Proposed Task Scheduling Algorithms

3.4.1 Simple greedy

The simple greedy algorithm only decides to compute a task based on the possibility of finishing it on time. In particular, the master calculates the estimated completion time of the most urgent task p (the task with the earliest deadline), using Eq. (3.5) and compares it with the deadline of this task. Assuming T_{current} is the current time

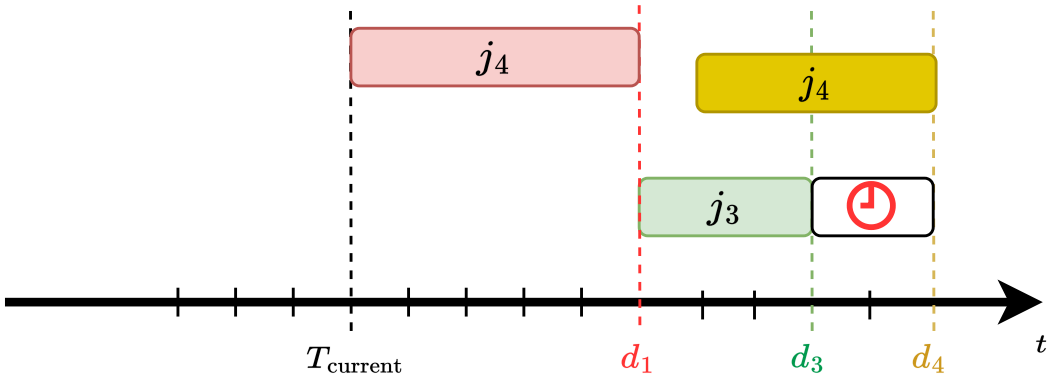

Figure 3.2: An example of the simple greedy scheduling algorithm

in the system, if:

$$\tilde{T}_{comp} = T_{\text{current}} + \frac{L_p}{Nr}[\alpha - \frac{1}{\mu}\log(1-r)] \tag{3.6}$$

is greater than d_p, the master ignores p. Otherwise, it examines the number of potential jobs finished before their deadlines in two scenarios: (i) p is dropped and (ii) p is sent to workers to be computed. To do so, it applies Eq. (3.6) for each existing task in these two scenarios. Now, only if the number of jobs (including p) in the second case is larger than the first case, the master sends p to the workers to be computed. Please note that since this algorithm only checks the most urgent task, and makes a decision about its execution by examining all arrived tasks, M tasks, its complexity is $O(M \log M)$.

In the following example, we will see how this algorithm works. Consider the scheduling problem shown in Fig. 2.3. First, because j_1 has the earliest deadline, the master decides whether to compute it. Based on deadlines and computation times, if the master decides to compute this task, j_1 and j_3 can be computed potentially. Otherwise, if it decides to drop the task, j_2 and j_3 may be finished before their deadlines. Hence, the master send j_1 to the workers, and then it decides to compute j_3.

Now let us assume that at the time j_2 was being computed, task j_4 arrived at the system at $T_{\text{current}} + 6$ with an approximate computation time of 4 units of time.

Algorithm 1: Farsighted greedy

while $|J| > 0$ **do**
  Sort J based on the deadline in descending order
  $Q = \emptyset$, $S = \emptyset$
  Approximate T_{comp} for newly arrived jobs based on (3.5)
  for $i = m, m-1, \cdots, 1$ **do**
    if $i == 1$ **then**
      $T = d_i$
    else
      $T = d_i - d_{i-1}$
    end if
    $Q = Q + \{(t_i, d_i, l_i)\}$
    while $T > 0$ and $Q \neq \emptyset$ **do**
      $j =$ Index of the job with the smallest T_{comp} in Q
      if $T^j_{\text{comp}} \leq T$ **then**
        $T = T - T^j_{\text{comp}}$
        $S = S + \{j\}$
        $Q = Q - \{(t_j, d_j, l_j)\}$
      else
        $T^j_{\text{comp}} = T^j_{\text{comp}} - T$
        $T = 0$
      end if
    end while
  end for
  Sort S based on the deadline
  If any new tasks arrive, update J.
end while

However, since the master finishes j_3 at $T_{\text{current}} + 8$, it does not have enough time to finish j_4 before its deadline, so this task is missed.

3.4.2 Farsighted greedy

In contrast to simple greedy algorithm that lacks comprehensive planning on all arrived tasks, farsighted greedy does not focus solely on the most urgent task. In particular, simple greedy makes sure not to sacrifice more than one job with a later deadline when executing the most urgent task. However, farsighted greedy examines all arrived jobs to ensure that executing each job with deadline d_m in the final schedule sacrifices at most one job with a later deadline.

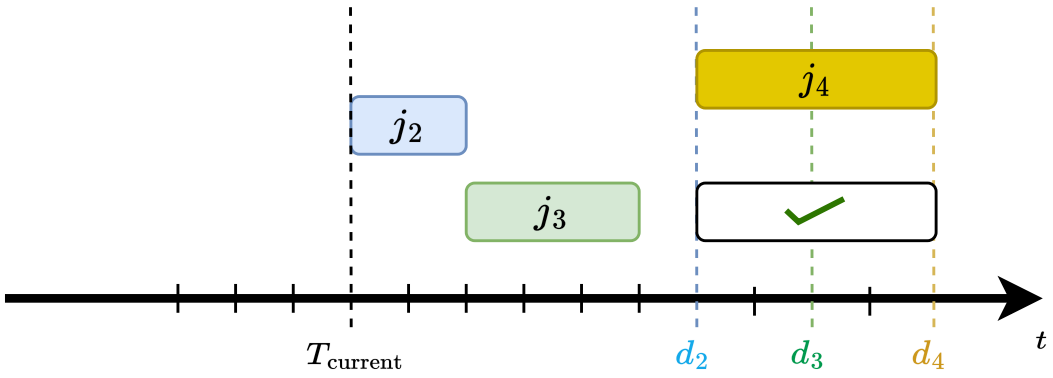

Figure 3.3: An example of the farsighted greedy scheduling algorithm

Also, farsighted greedy takes computation times into account when scheduling tasks. Hence, it makes its final schedule based on both deadline and computation time of tasks. The details of farsighted greedy, with time complexity of $O(M^2)$, is presented in Algorithm 1.

Considering the same example as in Section 3.4.1, we Assume there is a computation queue Q to which tasks are added in descending order based on their deadlines. First, task j_3 is added to Q. This algorithm determines what portion of j_3 can potentially be computed when the deadline of j_2 has already passed, i.e. $6 \leq T'_{\text{current}} \leq 8$. Next, j_2 is added to Q and the algorithm compares the computation time of this task with that of remained from j_3. Since the computation time of remaining portion of j_3 (1 unit of time) is shorter than the computation of time j_2 (2 units of time), the master decides to prioritize the computation of the remaining portion of j_3 in the interval $(d_1, d_2) = (5, 6)$. Since only one unit of time is available in that interval, the master can only finish j_3. Hence, task j_3 is send to the final schedule S and removed from Q. Then, j_1 is added to Q. As j_2 has a shorter computation time than j_1, the master prioritize the computation of j_2 in the interval $(T_{\text{current}}, d_1)$. Since after computation of j_2 there is only 3 units of time left to finish j_1, the master decides to drop task j_1 and sends task j_2 to the final schedule. Now the final schedule is sorted based on the deadlines, and j_2 and then j_3 are sent to the workers.

As illustrated in Fig 3.3, task j_4 arrives at the system at $T_{\text{current}} + 6$. Unlike simple

greedy that finishes j_3 at $T_{\text{current}} + 6$, since Farsighted greedy computes j_2 rather than j_1 before j_3, it will have enough time to finish task j_4 before its deadline. Thus, in this scenario Farsighted greedy computes tasks j_2, j_3 and j_4, while Simple greedy computes j_1 and j_3.

3.5 Simulation Results

In this section, we evaluate our proposed algorithms under various settings. We compare our algorithms with the non-preemptive shortest job first (SJF) algorithm, and SJF with dropping. In SJF with dropping, similar to the ordinary SJF algorithm, arrived jobs are sorted based on their expected computation times. However, the job with shortest computation time would be dropped if $\tilde{T}_{comp}$, calculated using (3.5), is greater than the job's deadline. We also use a genie-aided algorithm as a benchmark to compare our algorithms against an optimal, yet unfeasible, task scheduler. Assuming the total number of $M' > M$ jobs received in one scheduling simulation, the complexity of the genie-aided algorithm is $O(M'^2)$.

We assume $\alpha = \frac{1}{\mu} = 10^{-4}[s]$ and $N = 1000$ workers are available. In each iteration, the load of jobs are sampled uniformly from the interval $[1 \times 10^4, 2 \times 10^4]$. To generate task deadlines for our simulations, we first approximate the execution time of each task, using (3.5) with $r = 0.5$ and $\alpha = \frac{1}{\mu} = 10^{-4}[s]$, then multiply it by a value greater than one. By doing so, it is ensured that regardless of the arrival of other tasks and the way they affect the computation of one another, completion time of a given task is not longer than its deadline. Otherwise, some tasks would be impossible to complete. Note that $r = 0.5$ is solely used for generating task deadlines, while workers may employ a different coding rate to execute tasks.

Also, the average time gap between each task arrival, $\frac{1}{\lambda}$ (see (3.5)), is assumed to be around $2.54 \times 10^{-3}[s]$. This parameter, controls the frequency of job arrival in the system. We run Monte Carlo simulation for 10^5 iterations and terminate each iteration as soon as 10 computational jobs arrive at the system.

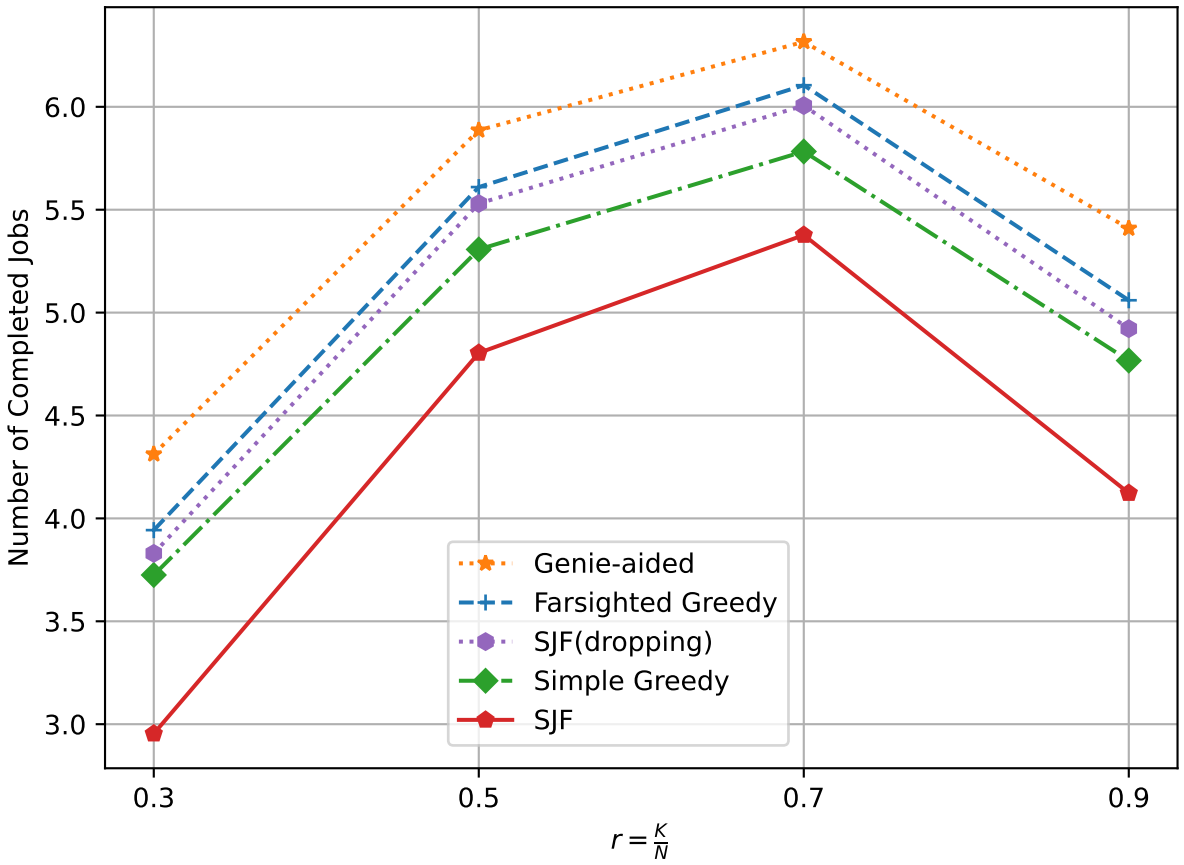

Figure 3.4: Average number of completed jobs vs r.

Fig. 3.4 illustrates the average number of completed jobs vs different code rates. The number of original sub-matrices K can be easily found as $K = N \times r$, where $N = 1000$. As seen simple greedy has completed more jobs than SJF which prioritizes the shortest job instead of the most urgent one. However, as SJF with dropping does not compute the shortest job at all costs, it supersedes simple greedy. Unlike simple greedy and SJF variants which mainly focus on either deadlines or computation times, farsighted greedy considers both the deadline and computation time of all arrived tasks to create a schedule. Hence, it has completed more jobs than simple greedy and SJF algorithms. Also, its performance is very close to the ultimate performance possible, i.e., that of the genie-aided algorithm.

Fig. 3.5 demonstrates wasteful operations per each completed job vs different code rates. We define an operation as the inner product of two vectors with a fixed num-

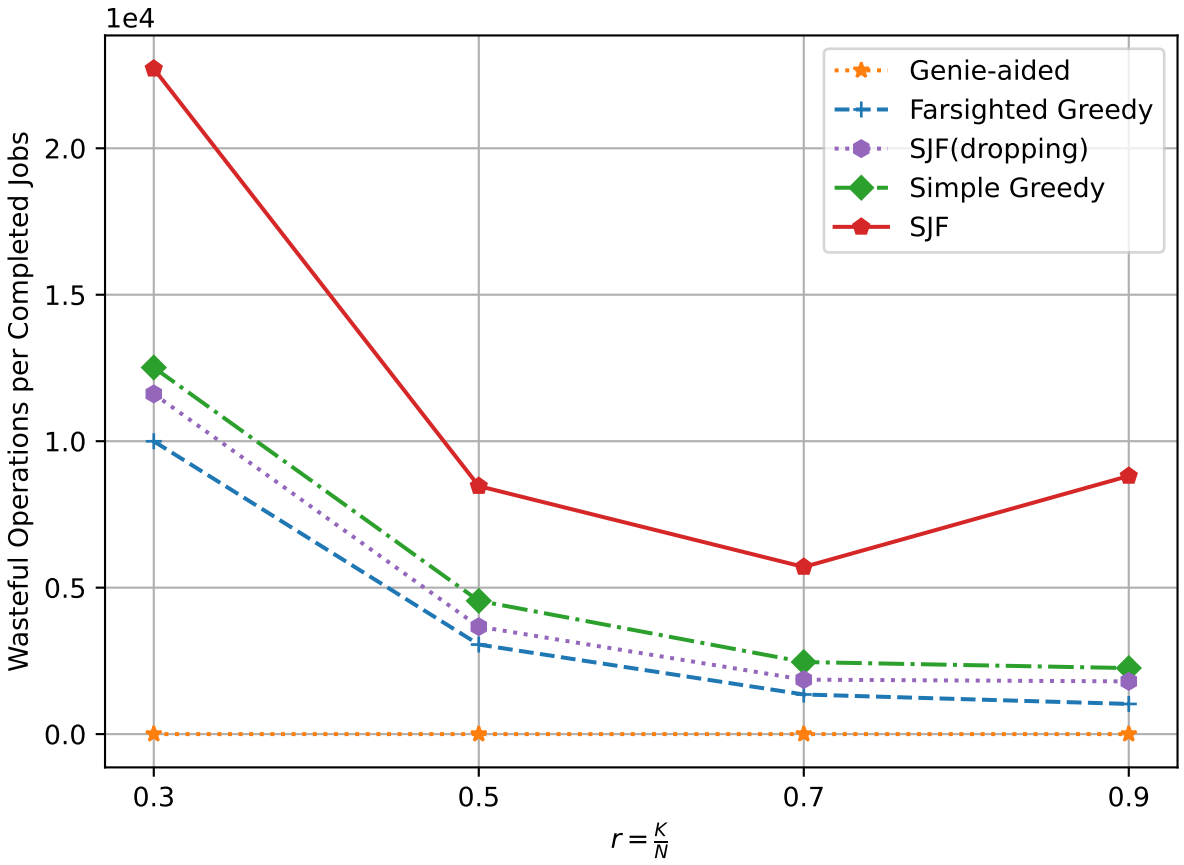

Figure 3.5: Average number of wasteful operations per completed jobs vs r.

ber of elements. A wasteful operation is one performed by a worker that does not contribute to the timely completion of the task. Specifically, when a worker does not finish its sub-task by the deadline, we consider the portion of sub-task which is finished before the deadline as wasteful operations. Also, when a task misses its deadline, even the operations done by workers which finished their sub-task on-time are considered wasteful, since they did not result in completion of the task before the deadline. The genie-aided algorithm does not perform any wasteful operations as expected. Furthermore, farsighted greedy has performed less wasteful operations than simple greedy and SJF algorithms at all code rates.

3.6 Conclusion

In this chapter, we considered the problem of task scheduling in a homogeneous distributed computing system, where each task has its execution deadline. MDS coding was adopted to alleviate the straggler effect. We proposed two greedy algorithms to maximize the number of completed tasks before their deadlines. Simulations showed that our proposed algorithms, with no knowledge of the exact execution time of tasks or the arrival time of future tasks, have a similar performance to a genie-aided algorithm that has access to this information. Our results further demonstrated that our algorithms have performed less number of wasteful operations compared to a simple SJF algorithm which schedules the job with shortest computation time first.

Chapter 4

Worker Assignment in Deadline-aware Heterogeneous Distributed Computing Systems

4.1 Introduction

In addition to time-critical computations, another significant aspect to be taken into account in distributed computing systems is the incorporation of tiered services. Cloud service providers sometimes offer different levels of services depending on their users' subscription tiers [62].

By offering a variety of service levels based on subscription tiers, these providers can accommodate users with different requirements and budget constraints. For instance, users seeking extensive storage capacity and significant processing power might opt for a higher subscription tier, granting them access to more abundant resources and enhanced performance. In such cases, these users expect expeditious responses and seamless execution of their tasks.

In addition to providing differentiated service levels, service providers acknowledge the significance of user satisfaction and strive to maintain a high level of customer experience. When an unfortunate event occurs, such as system downtime or service disruptions, and a subscriber's expectations are not met, the provider is obligated to address the issue. Depending on the user's subscription level, appropriate compensa-

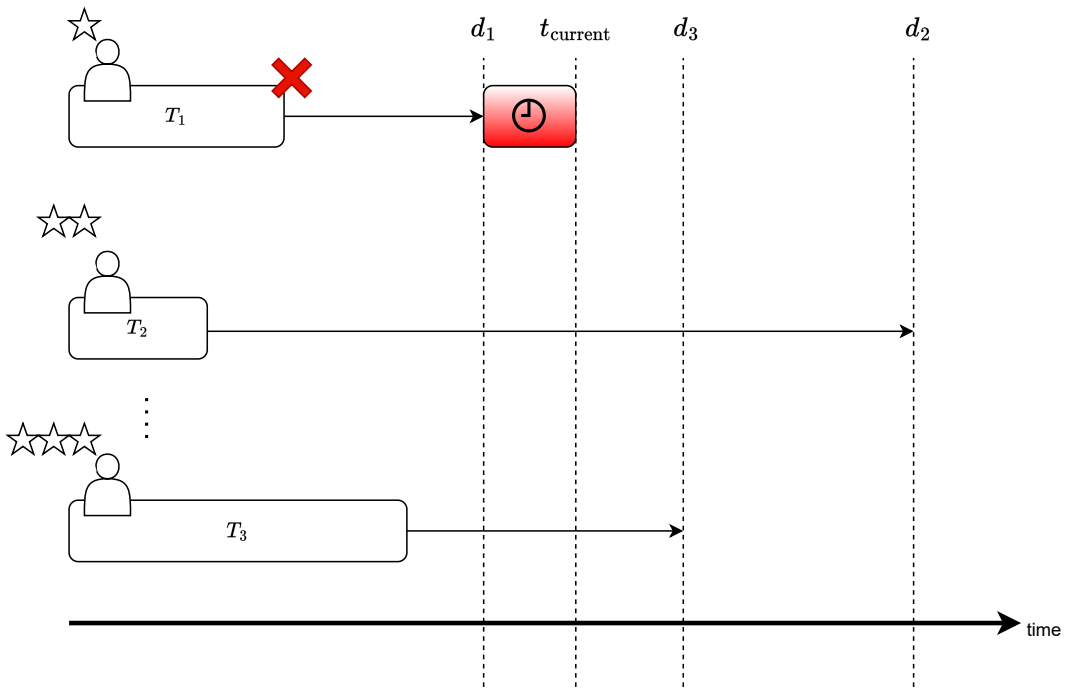

Figure 4.1: Time-critical computational tasks with different subscription classes

tion should be offered to rectify the inconvenience caused.

The tiered subscription model offers a win-win situation, as it enables distributed computing service providers to efficiently manage their resources while meeting the unique demands of various user segments. Users, on the other hand, can select a subscription tier that aligns with their specific needs and financial capabilities, making the cloud computing environment highly customizable and cost-effective.

In this chapter, we consider a heterogeneous CDC system in which we design a worker assignment scheme in order to maximize the reward the system receives. In particular, we consider multiple workers with different computational and communication capabilities. Furthermore, we assume there are several users who access this system at the same time. Each user has a matrix-vector multiplication task with a specific deadline. Additionally, each user has a subscription class and expects a certain level of service from the system according to its class. The system is rewarded by the uses, for being on-time, i.e. completing the task before its deadline, and based on the user's subscription level. As a result, often the system has to choose between a

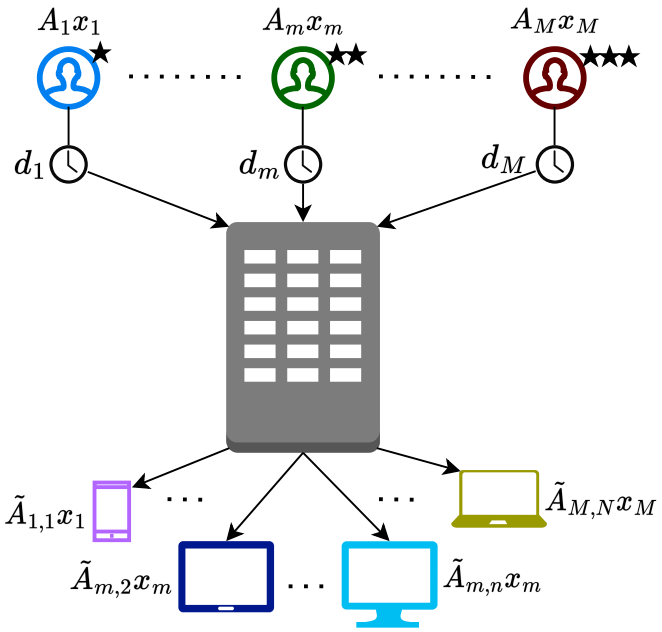

Figure 4.2: Illustration of a tiered distributed computing system with multiple time-critical tasks

task with a higher subscription level and one with an earlier deadline. To the best of our knowledge, this is the first work that considers both subscription class and task deadline in a CDC system.

To maximize the system reward, we present a greedy worker assignment algorithm, namely "reward greedy". Assuming M users are waiting for the result of their tasks and there are N workers at the system, the complexity of our algorithm is $O(MN)$. We compare our algorithm with the ultimate upper bound, that is a brute-force algorithm which exhaustively examines all possible worker assignment combinations. Simulation results show that our proposed algorithm provides about the same performance as the brute-force algorithm with much smaller complexity.

4.2 System Model

4.2.1 System architecture

As illustrated in Fig. 4.2 we consider a heterogeneous distributed computing system with N workers and M users, denoted by $\mathcal{N} = \{1, 2, \cdots, N\}$ and $\mathcal{M} = \{1, 2, \cdots, M\}$,

respectively. Also, there is a central controller that collects the computation and communication parameters of each worker along with information about deadlines, classes, and workloads of the users. Each user has a matrix-vector multiplication task $A_m x_m$ with the deadline d_m, where $A_m \in \mathbb{R}^{L_m \times S_m}$ and $x_m \in \mathbb{R}^{S_m}$. Thus, the task load, the required number of inner products of the task of user m, is L_m. Furthermore, we assume users are being served by different priorities. In particular, we define a three-level subscription class, $c_m \in \{1, 2, 3\}$. Each user belongs to one of the classes and users with higher values of c_m are prioritized for receiving services.

To reduce the straggler effect, we employ maximum-distance separable (MDS) codes. To achieve this, each user applies a MDS code to the rows of A_m to get its coded version $\tilde{A}_m \in \mathbb{R}^{\tilde{L}_m \times S_m}$, where $\tilde{L}_m \geq L_m$ denotes the number of coded rows. Then the encoded matrix $\tilde{\mathbf{A}}_m$ is horizontally divided into $N+1$ disjoint sub-matrices, $\tilde{A}_{m,0}$, $\tilde{A}_{m,1}$, $\cdots$, $\tilde{A}_{m,N}$, where $\tilde{A}_{m,n} \in \mathbb{R}^{l_{m,n} \times S_m}$. Note that $l_{m,0}$ represents the local computation workload at user m, while $l_{m,n}$, $\forall n \in \mathcal{N}$, denotes the m-th task assigned to the n-th worker. If $l_{m,n} = 0$, it indicates that the n-th worker is not assigned any workload from user m.

Assuming that worker n is assigned to user m, after the task is encoded and assigned to a group of workers, the user m sends the matrix $\tilde{A}_{m,n}$ to the worker n through their communication channel. As the code is MDS, each user m can recover its final result once it receives L_m inner products.

4.2.2 Computation and communication delay

We assume the communication delay largely arises from the delay in transmitting the coded sub-matrices to each worker, and transmitting x_m to the workers and the results to the users have negligible delay [31]. This is because the size of x_m and the result vector are much smaller than that of $\tilde{A}_{m,n}$. Hence, we consider the communication delay of transmitting $\tilde{A}_{m,n}$ to worker n, $T_{m,n}^{\text{comm}}$, follows an exponential distribution

[31, 63, 64] with the cumulative distribution function (CDF) given by

$$\mathbb{P}(T_{m,n}^{\text{comm}} \leq t) = \begin{cases} 1 - e^{-\frac{\gamma_n}{l_{m,n}}t}, & t \geq 0, \\ 0, & \text{otherwise}, \end{cases} \tag{4.1}$$

where $\frac{1}{\gamma_n}$ is the average delay of transmitting a single coded row to worker n from any user. Note that $T_{m,0}^{\text{comm}} = 0$ as local computation dose not need any communication.

Similar to Sec. 3.2.2, it is assumed that the time required by the worker n to compute l inner products is a shifted exponential random variable. Hence, the probability that the worker n finishes the computing of the workload $l_{m,n}$ before time t is

$$\mathbb{P}(T_{m,n}^{\text{comp}} \leq t) = \begin{cases} 1 - e^{-\frac{u_n}{l_{m,n}}(t - \alpha_n l_{m,n})}, & t \geq \alpha_n l_{m,n}, \\ 0, & \text{otherwise}. \end{cases} \tag{4.2}$$

In (4.2), α_n is the shift parameter at worker n, and u_n models the straggling effect at this worker.

Considering that $T_{m,n}^{\text{comp}}$ and $T_{m,n}^{\text{comm}}$ are two independent random variables, the total amount of time that the user m should wait to receive the sub-task assigned to worker n is $T_{m,n} = T_{m,n}^{\text{comp}} + T_{m,n}^{\text{comm}}$ with the CDF as follows [31]:

- if $\gamma_n \neq u_n$ and $t \geq \alpha_n l_{m,n}$:

$$\begin{aligned} \mathbb{P}(T_{m,n} \leq t) = 1 &- \frac{\gamma_n}{\gamma_n - u_n} e^{-\frac{u_n}{l_{m,n}}(t - \alpha_n l_{m,n})} \\ &+ \frac{u_n}{\gamma_n - u_n} e^{-\frac{\gamma_n}{l_{m,n}}(t - \alpha_n l_{m,n})} \end{aligned} \tag{4.3}$$

- if $\gamma_n = u_n$ and $t \geq \alpha_n l_{m,n}$:

$$\begin{aligned} \mathbb{P}(T_{m,n} \leq t) = 1 - & \\ & \left[1 + \frac{u_n}{l_{m,n}}(t - \alpha_n l_{m,n})\right] e^{-\frac{u_n}{l_{m,n}}(t - \alpha_n l_{m,n})} \end{aligned} \tag{4.4}$$

- In case $t \leq \alpha_n l_{m,n}$, then $\mathbb{P}(T_{m,n} \leq t) = 0$.

4.3 Problem Definition

Our goal is to design a joint worker assignment and load allocation policy that maximizes the system's benefit while satisfying all the users. The system's benefit is the

total reward it collects from the subscribed users. The worker assignment policy assigns workers to the users based on the users subscription class, the task deadline and the task load. Once the worker n is assigned to user m, it dedicates all its processing power to that user. Thus, we define the binary variable $k_{m,n}$ to indicate worker assignment. In particular, $k_{m,n} = 1$ indicates that worker n is assigned to user m and $k_{m,n} = 0$ means worker n does not serve user m.

In the rest of this section we first present our reward function and then formalize our optimization objective.

4.3.1 Task completion reward

As described in Section 4.2, each user m in the system has a specific deadline d_m for their task. However, the system does not necessarily drop a task if it exceeds the deadline. To incentivize the system to complete tasks before their deadlines, we introduce a penalty for unfinished computations that pass the deadline. Conversely, the system is rewarded for completing tasks before their respective deadlines. It is important to note that the reward for completing a task before the deadline is not equivalent to the penalty for completing it after the deadline with the same time offset. Our primary objective is to ensure timely completion of tasks rather than emphasizing how quickly they are finished.

In addition to the deadline, the system's reward is also influenced by the subscription class of each user, c_m. Specifically, completing a task for a user with a higher subscription class yields significantly greater rewards.

To meet the aforementioned requirements, we model the reward received by the system after completing task m in t_m units of time as follows:

$$R_m(t) = 1 - e^{-c_m(d_m - t_m)} \tag{4.5}$$

Let us illustrate an example to visually explain Eq. 4.5. As shown in Fig. 4.3, there are two tasks with the same deadline of 4 units of time. However, task 1, depicted as

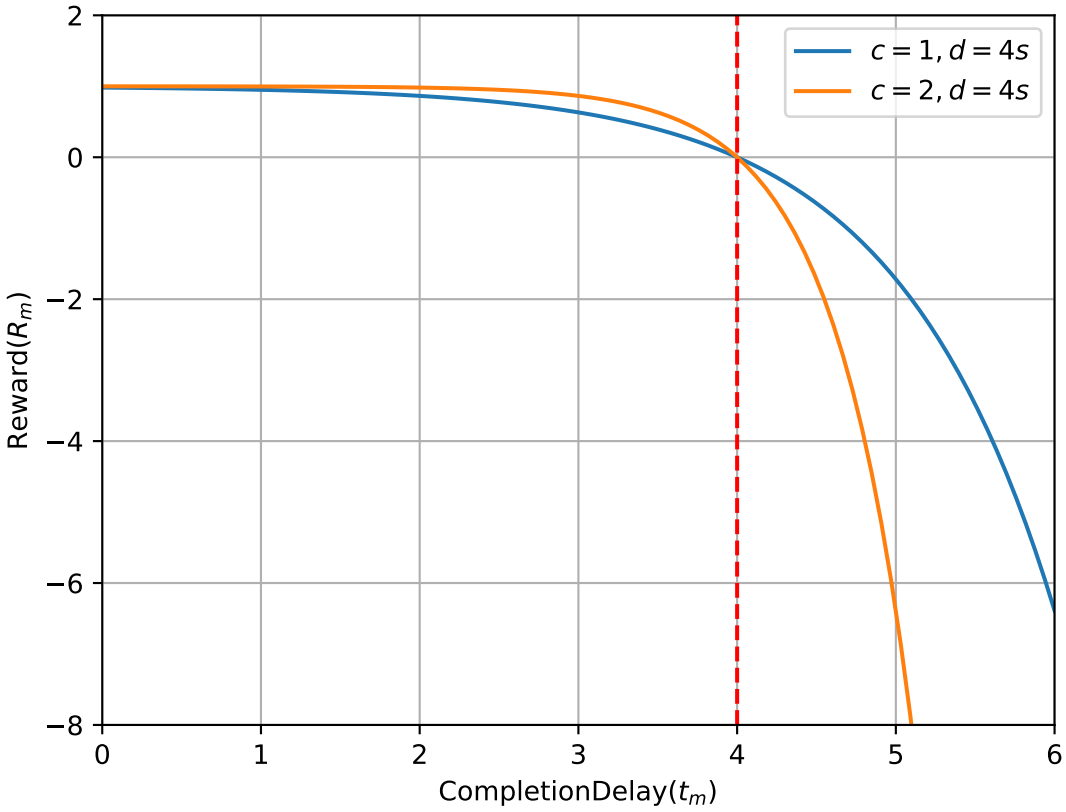

Figure 4.3: Percentage of overdue tasks for each user

orange, belongs to a higher subscription class ($c_1 = 2$) compared to task 2 ($c_2 = 1$) which is shown in blue. As depicted, the system is severely punished in case that each task is completed after the deadline, which is shown as the vertical red dashed line. Furthermore, the reward the system receives for completing task 1 before its deadline is higher than the reward it receives for completing task 2. Likewise, when task 1 is overdue, the system is penalized more than when task 2 is overdue.

4.3.2 Optimization objective

Our objective is to assign workers to the users and then optimally allocate workloads $l_{m,n}$ to the assigned workers such that the overall reward received by the system is

maximized. Hence, our optimization problem can be mathematically written as

$$\mathcal{P}_1 : \max_{\{l_{m,n},k_{m,n},t_m\}} \sum_{m=1}^{M} R_m = \sum_{m=1}^{M} 1 - e^{-c_m(d_m - t_m)} \tag{4.6a}$$

$$\text{s.t.} \quad \mathbb{E}[X_m(t_m)] \geq L_m, \, \forall m \in M, \tag{4.6b}$$

$$l_{m,n} \geq 0, \, \forall n \in N, \, \forall m \in M, \tag{4.6c}$$

$$k_{m,n} \in \{0, 1\}, \, \forall n \in N, \, \forall m \in M, \tag{4.6d}$$

$$\sum_{m=1}^{M} k_{m,n} = 1, \, \forall n \in N. \tag{4.6e}$$

Constraint (4.6b) guarantees that the average number of coded rows collected from workers until time t_m, $X_m(t_m)$, is sufficient to recover the final result. Although in practice the number of inner products assigned to each worker is an integer number, in (4.6c) this condition is relaxed for the sake of simplicity, where the only necessary condition for $l_{m,n}$ is to be a positive real number. The values obtained for $l_{m,n}$ will, however, be rounded up to the nearest integer after solving the optimization problem. Constraint (4.6d) simply suggests that whether worker n is serving user m. Also in constraint (4.6e), we force each worker to serve one and only one user.

Since the workers computations are independent, $\mathbb{E}[X_m(t_m)]$ can be written as [31]

$$\mathbb{E}[X_m(t_m)] = \sum_{n=1}^{N} l_{m,n} \mathbb{P}(T_{m,n} \leq t_m) \tag{4.7}$$

Based on Eq. (4.3) and Eq. (4.4), it can be shown that $\mathbb{P}(T_{m,n} \leq t_m)$ is a non-convex function. Therefore, using the Markov's inequality we approximate $E[X_m(t_m)]$ as

$$\mathbb{P}(T_{m,n} \geq t_m) \leq \frac{\mathbb{E}[t_m]}{t_m} = \frac{\theta_n l_{m,n}}{t_m}, \tag{4.8}$$

where $\theta_n \triangleq \frac{1}{u_n} + \frac{1}{\gamma_n} + \alpha_n$ indicates the average delay in communicating a single coded row to worker n and then computing the associated inner product at that worker. Hence, $\theta_n l_{m,n}$ is the average communication and computation delay of a sub-task with the load $l_{m,n}$. Furthermore, the lower bound for $\mathbb{P}(T_{m,n} \leq t_m) = 1 - \mathbb{P}(T_{m,n} \geq t_m)$

can be written as

$$\mathbb{P}(T_{m,n} \leq t_m) \geq (1 - \frac{\theta_n l_{m,n}}{t_{m,n}}). \tag{4.9}$$

Considering (4.9), we transform $\mathcal{P}_1$ into

$$\mathcal{P}_2 : \max_{\{l_{m,n}, t_m\}} \sum_{m=1}^{M} 1 - e^{-c_m(d_m - t_m)} \tag{4.10a}$$

$$\text{s.t.} \quad \sum_{n=1}^{N} l_{m,n}(1 - \frac{\theta_n l_{m,n}}{t_{m,n}}) \geq L_m, \, \forall m \in M \tag{4.10b}$$

$$\text{constraints (4.6c) and (4.6d) and (4.6e).} \tag{4.10c}$$

It can be simply verified that $l_{m,n}(1 - \frac{\theta_n l_{m,n}}{t_{m,n}})$ in (4.10b) is a convex function for each $m \in M$ and $n \in N$. Hence, (4.10b) which is the summation of convex functions, is convex itself. Nevertheless, due to the binary variable $k_{m,n}$, $\mathcal{P}_2$ is still a non-convex optimization problem.

Using (4.9), the load allocation problem for a single task, however, is a convex optimization problem. Assume the set of workers assigned to user m is already known and is denoted by Ω_m. Note that Ω_m also includes user m itself as it contributes to computation of its own task. The optimal load allocation policy for workers in Ω_m, which minimizes t_m, can be found by solving the following optimization problem

$$\mathcal{P}_3 : \min_{\{l_{m,n}, t_m\}} t_m \tag{4.11a}$$

$$\text{s.t.} \quad \sum_{n=1}^{N} l_{m,n}(1 - \frac{\theta_n l_{m,n}}{t_{m,n}}) \geq L_m, \, \forall m \in M, \tag{4.11b}$$

$$l_{m,n} > 0, \, \forall n \in \Omega_m, \, \forall m \in M. \tag{4.11c}$$

By solving $\mathcal{P}_3$, the optimal load allocation $l^*_{m,n}$ and its corresponding computation delay t^*_m are found as [31]

$$l^*_{m,n} = \frac{L_m}{\theta_{m,n} \sum_{n \in \Omega_m} \frac{1}{2\theta_{m,n}}}, \, \forall n \in \Omega_m, \, \forall m \in M \tag{4.12a}$$

$$t^*_m = \frac{L_m}{\sum_{n \in \Omega_m} \frac{1}{4\theta_{m,n}}}, \, \forall m \in M. \tag{4.12b}$$

The proof is provided in Appendix A.

Note that in Eq. (4.12a) and Eq. (4.12b), $\theta_{m,0} = \frac{1}{u_m} + \alpha_m$, where u_m and α_m are the computational properties of user m.

In the next section, we propose a greedy solution to $\mathcal{P}_2$, using Eq. (4.12a) and Eq. (4.12b).

4.4 Proposed solution

Algorithm 2: Reward greedy

Input: $\mathcal{N} = \{1, 2, \cdots, N\}$ and $\Omega_m = \{0\}$, $\forall m \in \mathcal{M}$
while $\mathcal{N} \neq \emptyset$ **do**
  $n^* = \arg\min_{n \in \mathcal{N}} \theta_n$.
  for $m \in \{1, 2, \cdots, M\}$ **do**
    Find $R_m(t_m^*; \Omega_m) = 1 - e^{-c_m(d_m - t_m^*)}$
    Find $R'_m(t_m^{*'}; \Omega_m + \{n^*\}) = 1 - e^{-c_m(d_m - t_m^{*'})}$
  end for
  $m^* = \arg\max_{m \in \mathcal{M}} R'_m - R_m$
  $\Omega_{m^*} = \Omega_{m^*} \cup \{n^*\}$, $\mathcal{N} = \mathcal{N} - \{n^*\}$
end while

In this section, we propose a solution called "reward greedy" for our worker assignment problem. As the name implies, the reward greedy algorithm assigns the best available worker to the user that would benefit the most from that worker, and continues this process until all workers have been assigned.

The reward greedy algorithm, with a time complexity of $O(MN)$, is presented in Algorithm 2. Initially, all workers are unoccupied, and we assume $\Omega_m = 0$ for all $m \in \mathcal{M}$, indicating that users perform tasks on their own without any assigned workers. The algorithm begins by identifying the best available worker n^*, which is the worker with the smallest θ_n among the workers not yet assigned to any user.

For each user m, the algorithm calculates t_m^* using Eq. (4.12b), which represents the current set of workers for that user, and determines the corresponding reward R_m. Next, assuming that worker n^* is added to Ω_m, the algorithm computes R'_m for each user m.

The user that would benefit the most from adding worker n^* to its set of workers is determined by finding the user with the largest difference between R'_m and R_m. This user is denoted as m^*, and worker n^* is assigned to it. These steps are repeated until all workers have been allocated.

4.5 Simulation Results

In this section, we present simulation results of our proposed algorithm in a coded distributed computing system. In order to evaluate our algorithm, we consider three benchmarks:

- *Uncoded computation with uniform worker assignment*: Users are assigned equal number of $\frac{N}{M}$ workers. Then each user m, equally divide A_m to $\frac{N}{M}+1$ sub-matrices ($\frac{N}{M}$ workers and user m itself) and distributes them to its workers.

- *Coded computation with uniform worker assignment*: Users are assigned equal number of $\frac{N}{M}$ workers. Then, the load allocated to each worker and to the user itself is calculated using Eq. (4.12a).

- *Brute-force search*: In this algorithm, all possible worker assignments are exhaustively searched. By calculating the potential reward could be obtained in each assignment, using Eq. (4.12a) and Eq. (4.12b), this algorithm chooses the assignment which leads to the maximum overall reward. Assuming there are M users and N workers, time complexity of this algorithm is $O(M^N)$, which makes this method impractical for real applications.

We consider a scenario where there are $M = 3$ users and $N = 7$ workers. User 1, 2 and 3 belong to subscription classes $c_1 = 1$, $c_2 = 2$ and $c_3 = 3$, respectively. The shift parameter α_n for each worker is randomly selected from $\{0.2, 0.25, 0.3\}$[ms]. Considering users have less computation power, their shift parameter is randomly chosen from $\{0.4, 0.5\}$[ms]. The rate parameter u_n is assumed to be $\frac{1}{\alpha_n}$, for all workers

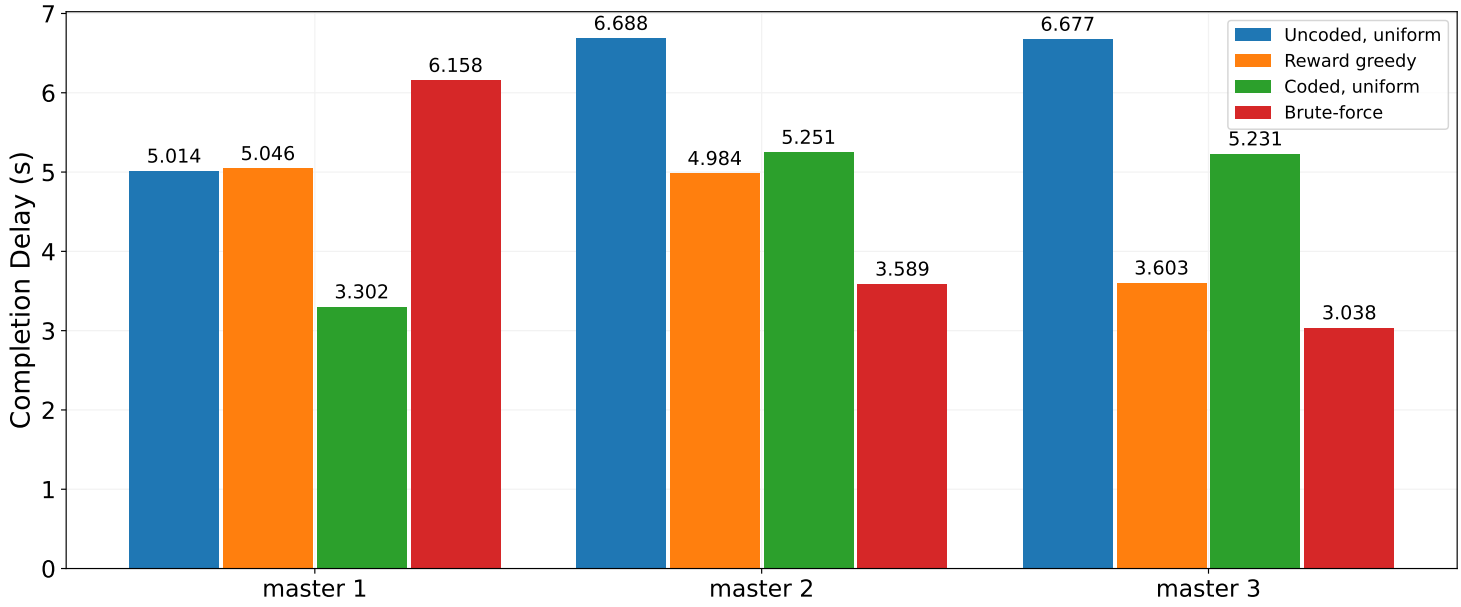

Figure 4.4: Average task completion delay

and users [31]. We also assume the communication ratio γ_n of each worker is $\gamma_n = 2u_n$. To fairly assign deadlines to tasks, we first calculate α_{average}, u_{average} and γ_{average} by taking the average of these parameters for all workers. Using these average parameters in Eq. (4.12b), we find the expected task completion delay for users assuming $\frac{N}{2}$ workers are assigned to each user. For each user, we assign the resulted time as the deadline d_m. Using the worker assignment schemes derived from the algorithms and the corresponding load allocation policies, we then run Monte Carlo simulation for 10^5 iterations.

Fig. 4.4 illustrates the average task completion delay for each user. It is important to note that the time complexity of algorithms is not taken into account in this figure. The completion delay represents the sum of computation time and communication time for each task. As depicted, both the reward greedy and brute-force algorithms finish user 3's tasks earlier compared to the other two users. This is because user 3 belongs to a higher subscription class than the other two users. However, the uncoded and coded uniform assignment algorithms do not consider the tasks' deadlines or their classes when allocating an equal number of workers to each user.

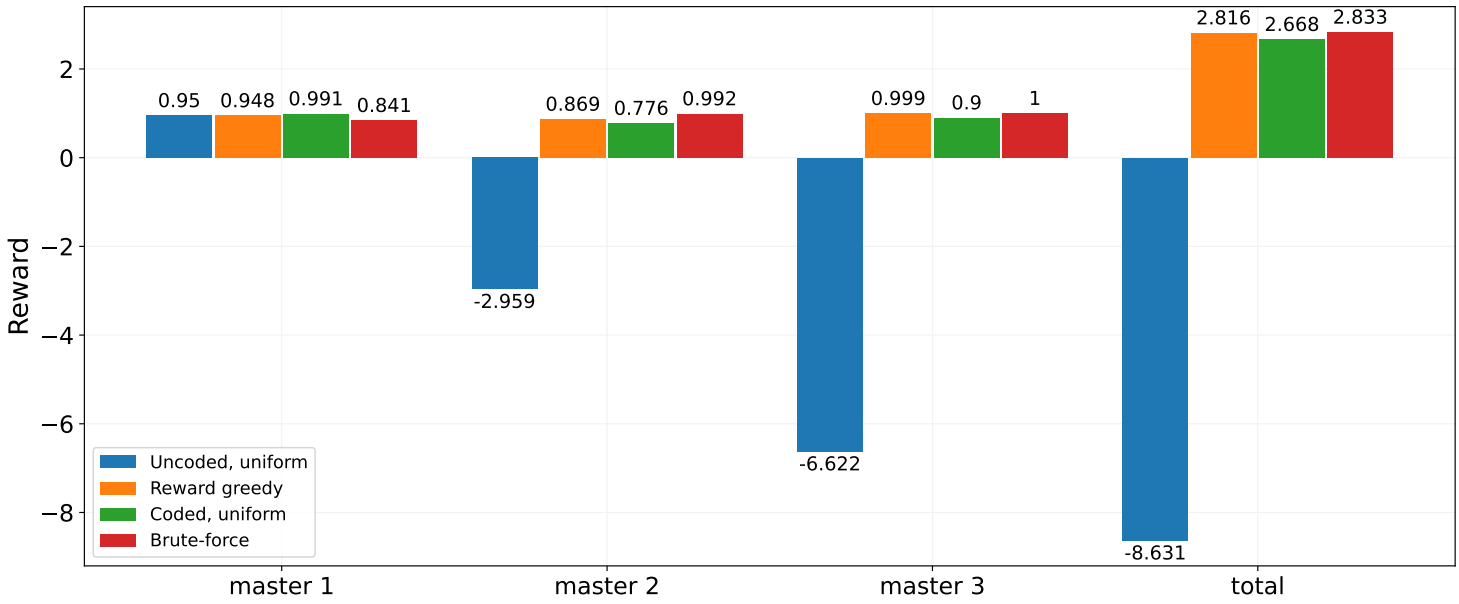

Figure 4.5: Average task completion delay

In Fig. 4.5, we utilize the run-times from Fig. 4.4 to calculate the reward obtained by the system from each user. The reward greedy algorithm achieves a total reward of 2.816 from the three users, which is approximately equal to that of the brute-force search algorithm, 2.833. Although the total reward for the brute-force algorithm is higher than that of other assignment schemes, it does not necessarily mean that it receives more reward for each specific user.

For example, the uniform assignment schemes complete user 1's task earlier than both reward greedy and brute-force, resulting in a higher reward collected from user 1. However, the total reward they receive from all three users is still smaller compared to brute-force and reward greedy. Notably, the total reward obtained from the uncoded uniform worker assignment algorithm is significantly lower than the other three algorithms, emphasizing the significance of both coding and a proper worker assignment policy.

Fig. 4.6 shows the percentage of time that each user's task is completed after its deadline. As seen brute-force has completed user 1's task after its deadline in more than 20% of iterations. Uniform worker assignment schemes, however, have almost

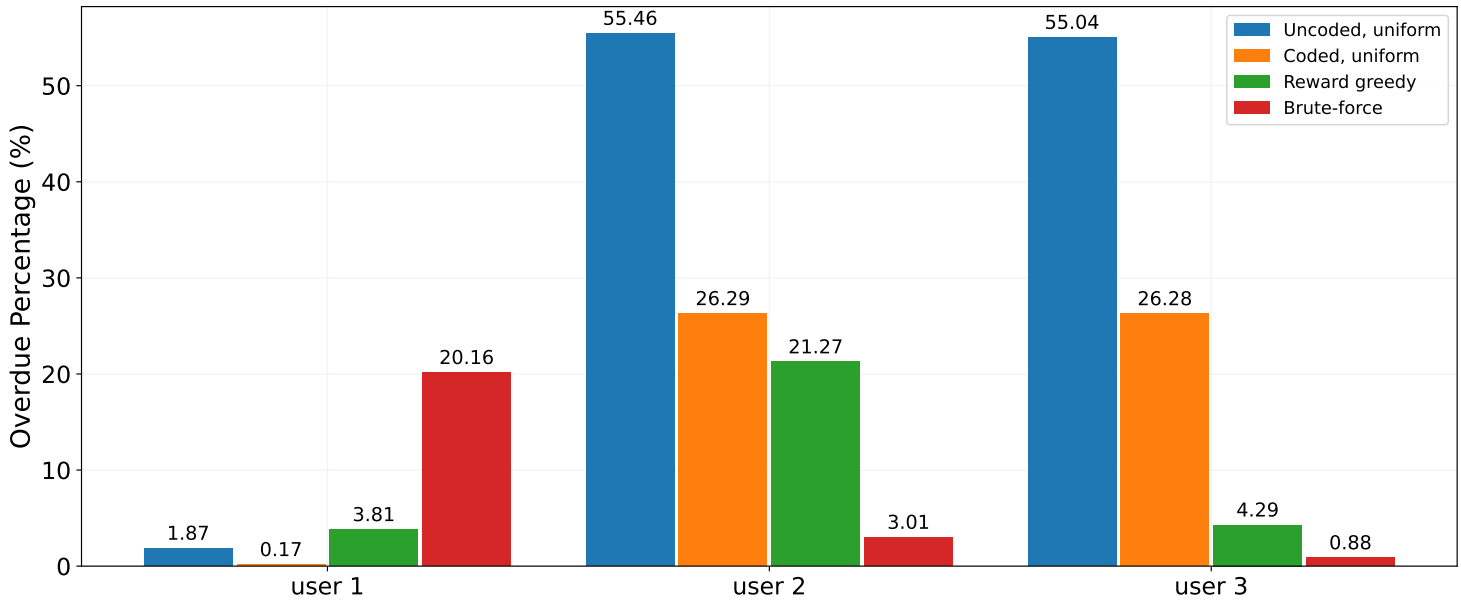

Figure 4.6: Percentage of overdue tasks for each user

always completed this task before its deadline. In contrast, reward greedy and brute-force perform better when it comes to tasks of users 2 and 3. In particular, performing uncoded uniform worker assignment, in more than 50% of iterations, these two tasks are completed after their deadlines.

4.6 Conclusion

We studied the problem of worker assignment in a heterogeneous coded distributed computing system. We consider a system consists of multiple users where each user belongs to a specific subscription class and has a time-sensitive matrix-vector multiplication task. To mitigate the negative impact of stragglers, MDS coding was adopted. We proposed a greedy worker assignment policy to maximize the total reward the system receives from users. According to our simulation results, our algorithm achieves a performance comparable to that of a brute-force search, but with significantly lower time complexity. Also, our results showed that by employing a proper coding scheme and a worker assignment policy the system reward significantly increases.

Chapter 5

Conclusions and Future Work

5.1 Summary of contributions and results

In this thesis, we studied coded distributed computing systems. By focusing on time-sensitive tasks, we aimed to address the practical needs of distributed computing environments, where timely and efficient execution is paramount. Furthermore, stragglers and inefficient resource allocation can lead to missed deadlines, degraded performance, and even system failures in these time-critical applications. Introducing time-sensitive tasks into the study allowed us to explore the impact of scheduling and worker assignment strategies on meeting deadlines and optimizing overall system performance.

The main contribution of our work lies in providing innovative and effective solutions to the challenges of task scheduling and worker assignment in coded distributed computing systems. Utilizing MDS coding and introducing greedy algorithms, we improve timely task completion in time-constrained environments.

In Chapter 3, we focused on task scheduling in a homogeneous distributed computing system, where each task had a specific execution deadline. The main challenges in such a system are random task arrivals and random execution times due to the straggling effect. To address these challenges, we proposed two task scheduling algorithms namely simple greedy and farsighted greedy and compared their performance with the ultimate upper bound, i.e., a genie-aided algorithm that knows the exact

arrival and execution times of all tasks.

In Chapter 4, we studied the worker assignment problem in a heterogeneous coded distributed computing system. We considered a system with multiple users, each with a time-critical task and belonging to a specific subscription class. Our objective was to devise a worker assignment policy that maximizes the overall reward of the system. To achieve this, we proposed a worker assignment policy called "reward greedy." Through simulation results, we demonstrated that our proposed algorithm achieves performance very close to that of a brute-force search while exhibiting significantly lower complexity.

5.2 Future research directions

As part of potential future research, an interesting avenue to explore is the study of fractional worker assignment policies. This approach involves allowing each worker to serve multiple masters simultaneously, effectively dividing its computational capability among different tasks. This opens up possibilities for more dynamic and flexible task allocation, where workers can handle multiple tasks concurrently, considering their capabilities and available resources.

Task scheduling in heterogeneous distributed computing environments presents another promising research area. In such environments, the computing resources available to the system exhibit significant diversity in terms of processing power, memory capacity, and network bandwidth. As a result, efficient and intelligent task scheduling becomes crucial to optimize the system's overall performance and resource utilization.

Another challenge arises when workers face memory constraints. Specifically, each worker is constrained by a maximum memory capacity, which limits the computation workload it can handle at any given time. In this practical scenario, the memory capacity of workers becomes a critical factor in load allocation. Hence, existing load allocation policies may no longer be applicable in such systems. One option is to de-

velop memory-aware allocation policies that take into account the memory constraints of workers and distribute computational loads accordingly.

Bibliography

[1] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao, "A survey on federated learning," *Knowledge-Based Systems*, vol. 216, p. 106 775, 2021.

[2] M. Ishaq, M. H. Afzal, S. Tahir, and K. Ullah, "A compact study of recent trends of challenges and opportunities in integrating internet of things (iot) and cloud computing," in *2021 International Conference on Computing, Electronic and Electrical Engineering (ICE Cube)*, 2021, pp. 1–4.

[3] N. T. Le, M. A. Hossain, A. Islam, D.-y. Kim, Y.-J. Choi, Y. M. Jang, *et al.*, "Survey of promising technologies for 5g networks," *Mobile information systems*, vol. 2016, 2016.

[4] B. Buyukates and S. Ulukus, "Timely updates in distributed computation systems with stragglers," in *2020 54th Asilomar Conference on Signals, Systems, and Computers*, 2020, pp. 418–422.

[5] C. McCann and J. Zahorjan, "Scheduling memory constrained jobs on distributed memory parallel computers," in *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, Ottawa, Ontario, Canada: Association for Computing Machinery, 1995, 208–219.

[6] A. Messer *et al.*, "Towards a distributed platform for resource-constrained devices," in *Proceedings 22nd International Conference on Distributed Computing Systems*, 2002, pp. 43–51.

[7] F. Cappello *et al.*, "Computing on large-scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid," *Future Generation Computer Systems*, vol. 21, no. 3, pp. 417–437, 2005.

[8] A. Iosup, O. Sonmez, S. Anoep, and D. Epema, "The performance of bags-of-tasks in large-scale distributed systems," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, ser. HPDC '08, Boston, MA, USA: Association for Computing Machinery, 2008, 97–108.

[9] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Comput. Surv.*, vol. 46, no. 4, 2014.

[10] K. Hwang, J. Dongarra, and G. C. Fox, *Distributed and cloud computing: from parallel processing to the internet of things*. Morgan kaufmann, 2013.

[11] A. Fernández *et al.*, "Big data with cloud computing: An insight on the computing environment, mapreduce, and programming frameworks," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 4, no. 5, pp. 380–409, 2014.

[12] G. Zhao, C. Ling, and D. Sun, "Sparksw: Scalable distributed computing system for large-scale biological sequence alignment," in *2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing*, IEEE, 2015, pp. 845–852.

[13] X. Li, L. Zhu, X. Chu, and H. Fu, "Edge computing-enabled wireless sensor networks for multiple data collection tasks in smart agriculture," *Journal of Sensors*, vol. 2020, pp. 1–9, 2020.

[14] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Computer Networks*, vol. 182, p. 107 496, 2020.

[15] K. Bilal and A. Erbad, "Edge computing for interactive media and video streaming," in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, IEEE, 2017, pp. 68–73.

[16] S. K. Rahimi and F. S. Haug, *Distributed database management systems: A Practical Approach*. John Wiley & Sons, 2010.

[17] W. Dai, H. Nishi, V. Vyatkin, V. Huang, Y. Shi, and X. Guan, "Industrial edge computing: Enabling embedded intelligence," *IEEE Industrial Electronics Magazine*, vol. 13, no. 4, pp. 48–56, 2019.

[18] X. Ma, H. Gao, H. Xu, and M. Bian, "An iot-based task scheduling optimization scheme considering the deadline and cost-aware scientific workflow for cloud computing," *EURASIP J. Wirel. Commun. Netw.*, vol. 2019, p. 249, 2019.

[19] J. Dean *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012.

[20] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, 74–80, 2013, ISSN: 0001-0782.

[21] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, 74–80, 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. [Online]. Available: https://doi.org/10.1145/2408776.2408794.

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

[23] D. Wang, G. Joshi, and G. Wornell, "Efficient task replication for fast response times in parallel computation," *SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, 599–600, 2014, ISSN: 0163-5999.

[24] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded mapreduce," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2015, pp. 964–971.

[25] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Fundamental tradeoff between computation and communication in distributed computing," in *2016 IEEE International Symposium on Information Theory (ISIT)*, 2016, pp. 1814–1818.

[26] Q. Yu, S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "How to optimally allocate resources for coded distributed computing?" In *IEEE International Conference on Communications (ICC)*, 2017, pp. 1–7.

[27] A. Reisizadeh, S. Prakash, R. Pedarsani, and S. Avestimehr, "Coded computation over heterogeneous clusters," in *2017 IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2408–2412.

[28] D. Kim, H. Park, D. Niyato, and J. Choi, "Worker assignment for multiple masters to speed up coded distributed computing in heterogeneous clusters," *IEEE Transactions on Services Computing*, pp. 1–16, 2022.

[29] F. Zhang, Y. Sun, and S. Zhou, "Coded computation over heterogeneous workers with random task arrivals," *IEEE Communications Letters*, vol. 25, no. 7, pp. 2338–2342, 2021.

[30] D. Kim, H. Park, and J. K. Choi, "Optimal load allocation for coded distributed computation in heterogeneous clusters," *IEEE Transactions on Communications*, vol. 69, no. 1, pp. 44–58, 2021.

[31] Y. Sun, F. Zhang, J. Zhao, S. Zhou, Z. Niu, and D. Gündüz, "Coded computation across shared heterogeneous workers with communication delay," *IEEE Transactions on Signal Processing*, vol. 70, pp. 3371–3385, 2022.

[32] Z. Liu, H. Zhang, B. Rao, and L. Wang, "A reinforcement learning based resource management approach for time-critical workloads in distributed computing environment," in *IEEE International Conference on Big Data (Big Data)*, 2018, pp. 252–261.

[33] V. Arabnejad, K. Bubendorfer, and B. Ng, "Scheduling deadline constrained scientific workflows on dynamically provisioned cloud resources," *Future Generation Computer Systems*, vol. 75, pp. 348–364, 2017, ISSN: 0167-739X.

[34] M. Hoseinnejhad and N. J. Navimipour, "Deadline constrained task scheduling in the cloud computing using a discrete firefly algorithm," *Int. J. Next Gener. Comput.*, vol. 8, 2017.

[35] C.-S. Yang, R. Pedarsani, and A. S. Avestimehr, "Timely-throughput optimal coded computing over cloud networks," in *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. Mobihoc '19, Catania, Italy: Association for Computing Machinery, 2019, 301–310, ISBN: 9781450367646.

[36] R. Singleton, "Maximum distanceq-nary codes," *IEEE Transactions on Information Theory*, vol. 10, no. 2, pp. 116–118, 1964.

[37] S. Kianidehkordi, N. Ferdinand, and S. C. Draper, "Hierarchical coded matrix multiplication," *IEEE Trans. Info. Theory*, vol. 67, no. 2, pp. 726–754, 2021.

[38] M. H. Ardakani, M. Mehrabi, M. Ardakani, and C. Tellambura, "On allocation of systematic blocks in coded distributed computing," *IEEE Communications Letters*, vol. 26, no. 4, pp. 748–752, 2022.

[39] M. A. Attia and R. Tandon, *Combating computational heterogeneity in large-scale distributed computing via work exchange*, 2017. arXiv: 1711.08452 [cs.DC].

[40] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments.," in *Osdi*, vol. 8, 2008, p. 7.

[41] J. S. Ng *et al.*, *A survey of coded distributed computing*, 2020. arXiv: 2008.09048 [cs.DC].

[42] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "A unified coding framework for distributed computing with straggling servers," in *IEEE Globecom Workshops*, 2016, pp. 1–6.

[43] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *IMach. Learn. Res.*, vol. 89, 2019, 1215–1225.

[44] M. Fahim and V. R. Cadambe, "Lagrange coded computing with sparsity constraints," in *Conference on Communication, Control, and Computing (Allerton)*, 2019, pp. 284–289.

[45] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial codes: An optimal design for high-dimensional coded matrix multiplication," in *Adv. Neural Inf. Process. Syst. (NIPS)*, 2017, 4403–4413.

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

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

[48] S. Dutta, V. Cadambe, and P. Grover, ""short-dot": Computing large linear transforms distributedly using coded short dot products," *IEEE Trans. Info. Theory*, vol. 65, no. 10, pp. 6171–6193, 2019.

[49] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. Avestimehr, "Coded computation over heterogeneous clusters," *IEEE Transactions on Information Theory*, vol. 65, no. 7, pp. 4227–4242, 2019.

[50] M. Kim, J.-y. Sohn, and J. Moon, "Coded matrix multiplication on a group-based model," in *IEEE International Symposium on Information Theory (ISIT)*, 2019, pp. 722–726.

[51] R. Tandon, Q. Lei, A. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *34th Int. Conf. Mach. Learn*, 2017, 3368–3376.

[52] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, "Tree gradient coding," in *IEEE International Symposium on Information Theory (ISIT)*, 2019, pp. 2808–2812.

[53] A. Mallick, M. Chaudhari, U. Sheth, G. Palanikumar, and G. Joshi, "Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication," in *of the ACM on Measurement and Analysis of Computing Sys.*, vol. 3, 2019.

[54] E. Ozfatura, S. Ulukus, and D. Gunduz, "Coded distributed computing with partial recovery," arXiv:2007.02191v1, 2020.

[55] A. Yazdanialahabadi and M. Ardakani, "A distributed low-complexity coding solution for large-scale distributed fft," *IEEE Transactions on Communications*, vol. 68, no. 11, pp. 6617–6628, 2020.

[56] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par'11, Bordeaux, France: Springer-Verlag, 2011, 90–109.

[57] A. B. Das and A. Ramamoorthy, "Distributed matrix-vector multiplication: A convolutional coding approach," in *2019 IEEE International Symposium on Information Theory (ISIT)*, 2019, pp. 3022–3026.

[58] K. Xiong and H. Perros, "Service performance and analysis in cloud computing," in *2009 Congress on Services - I*, 2009, pp. 693–700.

[59] B. Yang, F. Tan, Y.-S. Dai, and S. Guo, "Performance evaluation of cloud service considering fault recovery," in *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 571–576.

[60] J. Sgall, "Open problems in throughput scheduling," in *Algorithms – ESA 2012*, L. Epstein and P. Ferragina, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 2–11.

[61] D. Hyatt-Denesik, M. Rahgoshay, and M. R. Salavatipour, "Approximations for throughput maximization," *CoRR*, vol. abs/2001.10037, 2020.

[62] H. Goudarzi and M. Pedram, "Maximizing profit in cloud computing system via resource allocation," in *2011 31st International Conference on Distributed Computing Systems Workshops*, 2011, pp. 1–6.

[63] D.-J. Han, J.-y. Sohn, and J. Moon, "Coded distributed computing over packet erasure channels," in *2019 IEEE International Symposium on Information Theory (ISIT)*, 2019, pp. 717–721.

[64] D.-J. Han, J.-Y. Sohn, and J. Moon, "Coded wireless distributed computing with packet losses and retransmissions," *IEEE Transactions on Wireless Communications*, vol. 20, no. 12, pp. 8204–8217, 2021.

Appendix A: First Appendix

Since for $x > 0$ and $y > 0$, $f(x, y) = \frac{x^2}{y}$ is convex, $\mathcal{P}_3$ is a convex optimization problem. The Lagrangian of $\mathcal{P}_3$ is as follows:

$$\mathcal{L}(l_{m,n}, t_m, \lambda_m) = t_m + \lambda_m \left[L_m - \sum_{n \in \Omega_m} \left(l_{m,n} - \frac{\theta_{m,n} l_{m,n}^2}{t_m} \right) \right]$$

λ_m represents the Lagrange multiplier corresponding to (4.11b). The derivatives of $\mathcal{L}(l_{m,n}, t_m, \lambda_m)$ as follows:

$$\frac{\partial \mathcal{L}}{\partial l_{m,n}} = -\lambda_m + \lambda_m \theta_{m,n} \frac{2 l_{m,n}}{t_m} \tag{A.1a}$$

$$\frac{\partial \mathcal{L}}{\partial t_m} = 1 - \lambda_m \sum_{n \in \Omega_m} \frac{\theta_{m,n} l_{m,n}^2}{t_m^2} \tag{A.1b}$$

Also, the Karush-Kuhn-Tucker (KKT) conditions are as follows:

$$\frac{\partial \mathcal{L}_m}{\partial t_m^*} = 0, \ \frac{\partial \mathcal{L}_m}{\partial l_{m,n}^*} = 0, \forall n \in \Omega_m \tag{A.2a}$$

$$\lambda_m^* \left[L_m - \sum_{n \in \Omega_m} \left(l_{m,n}^* - \frac{\theta_{m,n} {l^*}_{m,n}^2}{t_m^*} \right) \right] = 0 \tag{A.2b}$$

$$\lambda_m^* \geq 0, \ l_{m,n}^* \geq 0 \tag{A.2c}$$

Upon solving the KKT conditions, we obtain the optimal load allocation and task completion delay for $\mathcal{P}_3$, as demonstrated in Eq. (4.12a) and Eq. (4.12b).