ABSTRACT

Title of Dissertation: ON SCHEDULING AND COMMUNICATION ISSUES IN DATA CENTERS

Sheng Yang
Doctor of Philosophy, 2020

Dissertation Directed by: Professor Samir Khuller
Department of Computer Science

The proliferation of datacenters to handle the rapidly growing amount of data being managed in the cloud, necessitates the design, management and effective utilization of the thousands of machines that constitute a data center. Many modern big data applications require access to a large number of machines and datasets for training neural nets or for other big data processing.

In this thesis, we present research challenges and progress along two fronts. The first challenge addresses the need to schedule communication between machines in a much more effective manner, as several running applications compete for network bandwidth. We address a basic question known as coflow scheduling to optimize the weighted average completion time of tasks that are running across different machines in a datacenter and to effectively handle their communication needs. Sometimes, we are forced to distribute a task among multiple datacenters due to cost or legal reasons. For this case, we also study a related model that addresses communication needs of tasks that process data on multiple data centers and

handles communication requirements of such tasks across a wide area network with possibly widely varying bandwidth and network structures across different pairs of machines.

The second challenge is from a cloud user's perspective - since access to resources such as those provided by Amazon AWS can be expensive at scale, cloud computing providers often sell under utilized resources at a significant discount via a spot instance market. However, these instances are not dedicated and while they offer a cheaper alternative, there is a chance that the user's job will be interrupted to make room for higher priority tasks. Certain non-critical applications are not significantly impacted by delays due to interruptions, and we develop an initial framework to study some basic scheduling questions under this circumstance.

In all of these topics, the problems we study are NP-hard and our focus is on developing good approximation algorithms. In addition, while we attack these problems from a theoretical perspective, all the algorithms developed in this thesis are practical and efficient, and can be easily deployed in practice, some are already deployed.

ON SCHEDULING AND COMMUNICATION ISSUES IN DATA CENTERS

by

Sheng Yang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Professor Samir Khuller, Chair/Advisor
Professor Uzi Vishkin, Dean's Representative
Professor Mosharaf Chowdhury
Professor David Mount
Professor Neil Spring

© Copyright by
Sheng Yang
2020

Acknowledgments

First and foremost, I would like to thank Professor Samir Khuller for being the best advisor I could ever imagine. His incredible wisdom and ever-lasting energy never failed to impress me, not to mention his mastering in the art of explaining complicate stuff in a clean and insightful way. In spite of being the chair of the CS Department for the University of Maryland, College Park and then for the Northwestern University, Samir has always made himself available for help and advice, as a mentor and a friend. I shall see him as a role model for a researcher and an advisor.

Over the past few years, I have had the privilege of working with some great researchers during my summer internships. I am very grateful to my mentor Randeep Bhatia at Bell Labs, and Kanak Mahadik and David Jacobs at Adobe. The experience at Adobe led to further collaborations with Subrata Mitra, Sunav Choudhary, and Kanak Mahadik, which provides me a viewing angle from industry I would be otherwise unable to see. I was also lucky to work with Professor Mosharaf Chowdhury, who brought tremendous new ideas from the system community which made this thesis possible. I would like to thank Professor Uzi Vishkin, Professor Mosharaf Chowdhury, Professor David Mount, and Professor Neil Spring for agreeing to be on my thesis committee. Their questions and comments helped me reflect on my

research from different perspectives. And special regards to the entire staff of the Computer Science Department, in particular — Jennifer Story, Adelaide Findlay, Tom Hurst, Jodie Gray, and Sharron McElroy, for their warm help, especially those related to my visiting to Northwestern University.

I would like to thank Professor Jian Li, my undergraduate advisor, for introducing me to the world of theoretical computer science, and encouraging me to pursue a PhD. His suggestions have been lighthouses that give me guidance throughout the past few years.

My experience at UMD has been pleasant and unforgettable. I would like to thank all my friends: Alejandro, Shuhao, Pan, Yue, especially my lab mates Ahmed, Ioana, Saba, Pattara, Manish, with whom countless hours of discussions were made, which has been a pleasure and a rich source of fruitful outcomes. The life as an international student can be hard, but the hardness has been greatly reduced for all my Chinese friends that makes it feel like home: Jianxin, Liyan, Qiang, Honghao, Yiyang, Xin, Xiaodi, Penghui, Xingyao, Xuchen, Zeyu. Among them, the greatest thank is to Tongyang, with whom I shared apartment for my whole Maryland residency, in addition to the four years in college. He helped me for life and research alike, building up a life-long friendship. The last year and a half I was visiting Northwestern, where I met Professor Jason Hartline, Professor Konstantin Makarychev, Professor Aravindan Vijayaraghavan, together with their students and post-docs: Chenhao, Xue, Liren, Yingkai, Yiding, Yifan, Aidao, Hedyeh, Sumedha, Aravind, and Sanchit. The year 2020 has been hard for everyone, but their help and company has been the light in the darkness.

I would like to acknowledge financial support from NSF grants CNS 156019 and CCF 1655073 (Eager). Part of the research is supported by research grant from Amazon and Adobe. And special thanks to Northwestern University for supporting me as a visiting pre-doctoral scholar.

Finally, I awe my deepest thanks to my family, my fater Qichang Yang and mother Meiqin Zhang. None of this would have been possible without their love and constant encouragements.

Table of Contents

Acknowledgements ii

Table of Contents v

List of Figures viii

Chapter 1:Introduction 1

1.1 Coflow Scheduling in Switch Model 4
1.2 Coflow Scheduling in Networks 7
1.3 Scheduling with Spot Instance 8
1.4 Outline of the Dissertation 11

Chapter 2:Coflow Scheduling in Switch Model 13

2.1 Related Works 13
2.1.1 Relationship with Concurrent Open Shop 15
2.2 Our Contributions 17
2.3 Preliminaries 18
2.3.1 Scheduling a Single Coflow 19
2.3.2 Linear Programming Relaxation 20
2.4 High Level Ideas 21
2.5 Approximation Algorithm for Coflow Scheduling with Release Times 23
2.5.1 Finding a Permutation of Coflows Using a Primal Dual Algorithm 24
2.5.2 Scheduling Coflows According to a Permutation 26
2.6 Analysis 28
2.6.1 Coflows with Zero Release Times 28
2.6.2 Coflows with Arbitrary Release Times 30
2.6.3 Analyzing the Primal-Dual Algorithm 33
2.6.4 Primal Dual Analysis 34
2.7 An Alternative Approach Using LP Rounding 38
2.7.1 Proof of the LP Rounding Version of the Main Theorems . . . 39
2.8 A Combinatorial 3-approximation Algorithm For Concurrent Open Shop with Release Times 41
2.9 Correction of Algorithm by Qiu et al. 42
2.9.1 Interval-Indexed LP Formulation 42
2.9.2 Grouping Coflows 43

2.9.3 Error . . . 44
2.9.4 Corrected Grouping Algorithm . . . 45
2.10 Counterexample to Claim by Luo et al. . . . 48

Chapter 3: Coflow Scheduling in Networks **50**
3.1 Introduction . . . 50
3.1.1 Related Works . . . 53
3.1.2 Our Contributions . . . 54
3.1.3 Chapter Organization . . . 54
3.2 Model and Problem Definition . . . 55
3.3 Linear Programming Relaxation . . . 58
3.3.1 Model-specific Constraints . . . 60
3.4 Approximation Algorithms . . . 62
3.4.1 Stretch Algorithm . . . 63
3.4.2 Analysis . . . 65
3.5 Hardness of Approximation . . . 69
3.6 Experiments . . . 71
3.6.1 Implementation Details . . . 72
3.6.2 Baselines . . . 73
3.6.3 Experimental Results . . . 78
3.7 Sketch of generalization to super-polynomial time span . . . 79
3.7.1 Analysis . . . 81
3.8 Conclusion . . . 84

Chapter 4: Scheduling with Spot Instances **86**
4.1 Introduction . . . 86
4.1.1 Training ML jobs on the Cloud . . . 88
4.1.2 Our Contributions . . . 90
4.1.3 Our Techniques . . . 90
4.2 Problem Formulation . . . 91
4.2.1 Notations . . . 91
4.2.2 Scheduling with Spot Instances . . . 92
4.2.3 Final Problem Statement . . . 93
4.2.4 Eliminating an Assumption . . . 94
4.3 Continuous Optimization Phase . . . 96
4.3.1 Stochastic Knapsack Exponential Constraints . . . 96
4.3.2 Stochastic Knapsack Polynomial Constraints . . . 99
4.3.3 Construct a solution $\{z_{\pi,i,t}, y_{\pi,t}\}$ of ExpP from a solution $\{x_{u,t}, s_{u,t}\}$ 100
4.3.4 Continuous Optimization . . . 101
4.4 Rounding Phase . . . 102
4.4.1 Contention Resolution Scheme . . . 102
4.4.2 Rounding Algorithm . . . 103

Chapter 5: Conclusion **111**
5.1 Future Directions . . . 113

5.1.1 Coflow Scheduling 113
5.1.2 Scheduling Spot Instances 114

Bibliography **116**

List of Figures

1.1 An example coflow over a 2×2 switch. The figure illustrates two equivalent representations of a coflow - (i) as a weighted, bipartite graph over the set of ports, and (ii) as a $m \times m$ integer matrix. . . . 5

2.1 LP_1 for Coflow Scheduling 21
2.2 Example that illustrates sequentially scheduling coflows independently can lead to bad schedules. 22
2.3 Dual of LP_1 24
2.4 Simple counterexample to Claim 3 49

3.1 Example of coflow. The first graph shows the network topologies and the bandwidth of each link. We have one coflow consisting of two flows: one from NY to BA of demand 18 (denoted with dashed, green lines), the other from HK to FL of demand 12 (denoted with solid, red lines). The second graph shows the single path model, where each flow needs to be transmitted along a given path. It also implies a schedule in this model: transmit according to the path for 3 time units, and both flows are done. The third graph shows the free path model, where each flow can be split along multiple paths as long as the capacity of edges are respected. Here both flows can share the link from NY to FL and the entire coflow finishes in 2 units of time. . 53
3.2 On the left is the graph structure: bi-directed edge of independent capacity of 1, on the right is the demanded coflow. There are four coflows each containing one single flow: red (solid) from v_1 to t, green (dashed) from v_2 to t, orange (dotted) from v_3 to t, and blue (curly) from s to t. The first three have demand 1, while the blue coflow has a demand of 3. All of them have the same weight of 1. 57
3.3 For the *single path* model, we have the path assignment in the left figure. Notice the path for green (dashed) flow shares an edge with that for the blue (curly) flow. Here is one optimal solution for the *single path model*. The total weighted completion time is $1+1+1+4 = 7$. 57
3.4 This is the optimal solution in the *free path model*. At time 1, send the red (solid), green (dashed), and orange (dotted) coflows. At time 2, send the blue (curly) coflow on all paths. The total weighted completion time is $1 + 1 + 1 + 2 = 5$. 57

3.5 Here we show an example solution obtained from the LP, different color indicate different flows. In the second picture, we stretch with $\lambda = 0.5$. In the third picture, we leave the slots empty if the corresponding flow is finished. In the fourth picture, we utilize the idle slots and move some flows to earlier times. Though this does not improve the theoretically bound, it is beneficial in practice and is used in our experimental evaluation. 64
3.6 Free path model on SWAN, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. 75
3.7 Free path model on G-Scale, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. 75
3.8 Free path model on SWAN for workload FB, the different choice of time interval ϵ may affect the performance bound of time interval LP value and the performance of heuristic ($\lambda = 1$). 75
3.9 Single path model on SWAN, showing the performance bound of time indexed and time interval LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against algorithm by Jahanjou et al.[2]. 76
3.10 Single path model on G-Scale, showing the performance bound of time indexed and time interval LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against algorithm by Jahanjou et al.[2]. 76
3.11 Free path model with no weight on graph SWAN, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against Terra[17] 77
3.12 Free path model with no weight on graph G-Scale, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against Terra[17]. 77

Chapter 1: Introduction

With the slowing of Moore's law, significant new advances in improvement in processor clock speeds having slowed, researchers have turned to massively parallel machines to build significant processing capacity to successfully attack the next generation of data intensive computational problems. Modern computation facilities (datacenters) usually consist of hundreds to millions of machines that coordinately are capable of performing tasks that no single computer can accomplish. Such coordination calls for new frameworks that can effectively leverage the underlying hardware, upon which thousands if not millions of applications run.

One example of a powerful framework to enable processing in data centers would be Map-Reduce. In Map-Reduce, a computational task is broken into phases that involves mappers that distribute parallel processed data and reducers that collect and aggregate the output and pass them to the next phase. Each round involves a significant amount of communication of data between machines. This communication is so substantial that has become a major bottleneck in our ability to process large scale data quickly. Thus, we focus mainly on the communication phases that are involved. To address this challenge, a new scheduling problem called *coflow scheduling* was defined by Chowdhury et al. [1]. In this problem,

each application has a communication matrix D that specifies the communication required between each pair of machines. Each application can proceed to the next round when all of its communication needs are met. The goal is to minimize the weighted average completion time. In the original model, a data center is modeled as a giant switch, through which any pair of machines can communicate by sending a message, and we assume uniform bandwidth between different pairs of machines. In addition, we also study a related model recently proposed by Jahanjou et al. [2], which models the communication network as a general graph. This framework can also be used to model a heterogeneous network within a data center, or a network spanning multiple data centers. Certain applications are forced to run across data centers to access data that is geographically distributed, for cost or legal reasons. In this model, the application may send data along a single path or along multiple paths simultaneously. We also allow a machine to be communicating data to multiple other machines at the same time, provided the bandwidth constraints are not violated. Since most of the problems we study in this area generalize the classical *open concurrent shop problem* and are therefore NP-hard, our approach is to develop approximation algorithms and recent studies [3] have shown both the practicality and efficiency of the methods we develop.

Another problem we consider traces its root to the origin of "cloud computing". Large datacenters require immense investments to build, let alone the cost in electricity and salary for datacenter maintenance specialist. It would be a huge burden if every company with computing needs has to build and maintain its own datacenters, especially when the need is temporal. Therefore, there is great demand

for a cheaper access to computing power. In fact, such computing powers do exist. Big companies build datacenters to meet their peak needs, e.g. Amazon builds its datacenters to handle the extraordinary demand on Black Friday (Cyber Monday, Prime Day, etc.), which is several times that of an average day. However, such peak demands do not last long, leading to the poor utilization of datacenters off-peak. Demands drop, machines run idle, but maintenance cost remains inelastic, leading to unnecessary cost. The idea of "cloud computing" emerges when these companies started to rent their surplus computing power to smaller entities. Such computing power, utilizing the idle machines, incurs minimal marginal cost in the datacenters. In fact, the marginal cost of additional computing power for big companies is so low such that cloud services are no longer built purely on surplus computing power, but also on dedicated servers. While dedicated servers are used, the tension from fluctuating demands persists: to ensure the performance under peak demands, companies still reserve much more than they normally need, causing machines to idle most of the time. To improve utilization and help mitigate costs, major cloud providers offer a discounted option called *spot instance* (*Spot Instance* for Amazon AWS, *low-priority VMs* for Microsoft Azure and *preemptible instances* for Google Cloud). Fundamentally, one can purchase certain configurations at a steep discount, with one caveat — the jobs receive lower priority and may be interrupted.

This will be problematic for most jobs, but not a big hurdle for others, e.g. machine learning training jobs. Such jobs make checkpoints periodically and can recover from interruption. The main obstacle is the stochastic nature of the interruptions. We face the trade-off between low cost and uninterrupted computations.

We study the problem of scheduling with the presence of spot instances, maximize the total utility obtained under a given budget.

In this dissertation, we focus on algorithmic problems arising from different perspectives of cloud computing. In the following sections, we describe the problems we consider in detail.

1.1 Coflow Scheduling in Switch Model

Large scale data centers have emerged as the dominant form of computing infrastructure over the last decade. The success of data-parallel computing frameworks such as MapReduce [4], Hadoop [5], and Spark [6] has led to a proliferation of applications that are designed to alternate between computation and communication stages. Typically, the intermediate data generated by a computation stage needs to be transferred across different machines during a communication stage for further processing. For example, there is a "Shuffle" phase between every consecutive "Map" and "Reduce" phases in MapReduce. With an increasing reliance on parallelization, these communication stages are responsible for a large amount of data transfer in a datacenter. Chowdhury and Stoica [1] introduced coflows as an effective networking abstraction to represent the collective communication requirements of a job. We consider the problem of scheduling coflows to minimize weighted completion time and give improved approximation algorithms for this basic problem.

The communication phase for a typical application in a modern data center may contain hundreds of individual flow requests, and the phase ends only when

all of these flow requests are satisfied. A coflow is defined as the collection of these individual flow requests that all share a common performance goal. The underlying data center is modeled as a single $m \times m$ *non-blocking switch* that consists of m input ports and m output ports. We assume that each port has a unit capacity, i.e., it can handle at most one unit of data per unit time. Modeling the data center itself as a simple switch allows us to focus solely on the scheduling task instead of the problem of *routing* flows through the network. Each coflow j is represented as an $m \times m$ integer matrix $D^j = [d^j_{io}]$ where the entry d^j_{io} indicates the number of data units that must be transferred from input port i to output port o for coflow j. Figure 1.1 shows a single coflow over a 2×2 switch. For instance, the coflow depicted needs to transfer 2 units of data from input a to output b and 3 units of data from input a to output d. Each coflow j also has a weight w_j that indicates its relative importance and a release time r_j.

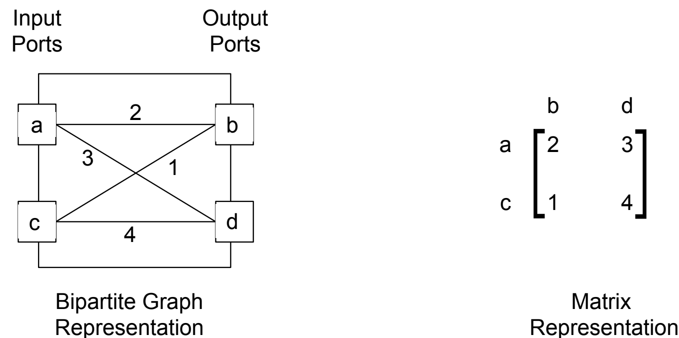

Figure 1.1: An example coflow over a 2×2 switch. The figure illustrates two equivalent representations of a coflow - (i) as a weighted, bipartite graph over the set of ports, and (ii) as a $m \times m$ integer matrix.

A coflow j is available to be scheduled at its release time r_j and is said to be completed when all the flows in the matrix D^j have been scheduled. More formally,

the completion time C_j of coflow j is defined as the earliest time such that for every input i and output o, d_{io}^{j} units of its data have been transferred from port i to port o. We assume that time is slotted and data transfer within the switch is instantaneous. Since each input port i can transmit at most one unit of data and each output port o can receive at most one unit of data in each time slot, a feasible schedule for a single time slot can be described as a matching. Our goal is to find a feasible schedule that minimizes the total weighted completion time of the coflows, i.e., minimize $\sum_j w_j C_j$. We aim to find approximation algorithm. Suppose an optimal solution can achieve weighted completion time OPT, and our algorithm produces a solution with a value SOL that satisfies SOL $\leq \alpha$OPT for any input, then we say our algorithm produces an α-approximation.

This problem was first introduced by Chowdhury and Stoica [1] to describe the prevalent communication patterns in data centers. It has been a hot topic in the systems community [7, 8, 9, 10, 11, 12]. In addition, it also caught interest of the theory community, with a line of work [2, 13, 14, 15, 16] leading to the state-of-the-art combinatorial approximation algorithm [15], which is the main content of Chapter 2. In fact, this theoretical result manages to make its way back to the system community, closing the loop of research from system to theory, and back to system. A system called Sincronia [3] was developed based on the primal-dual method. Their algorithm only allows zero release time, therefore looks simpler than the algorithm in Chapter 2 which supports arbitrary release times. This system improves upon state-of-the-art methods and gives practical and near-optimal solutions in real testbeds.

In Chapter 2, joint work with Saba Ahmadi, Samir Khuller, and Manish Purohit [15], we give a 4-approximation algorithm for the case without release time, and 5-approximation with release time. In addition to an LP-based solution, we also give a primal-dual based version with the same approximation bounds that is combinatorial and efficient in practice.

1.2 Coflow Scheduling in Networks

The switch model is limited to uniform capacities and topology. Over time, machines and links fail and get replaced or upgraded, resulting in non-uniformity. Geo-distributed networks experience even worse uniformity with various link types, evolving network structures, and unbalanced capacity. Direct links may be missing between certain pairs of nodes, that demands proper routing of data transmission.

In order to solve these problems, a slightly different model of coflow scheduling was proposed by Jahanjou et al. [2], which assumes that the underlying connection between machines is an arbitrary directed graph rather than a complete bipartite graph. Each node can be a machine, a datacenter or an exchange point (switch, router, etc.), and an edge between two nodes represents a physical link between the two Internet infrastructures. When some data needs to be transmitted from one node to another, it needs to be transmitted along a chain of edges. Unlike in the switch model where only one packet can be sent at each time slot, data for multiple jobs is allowed to transfer on the same link at the same time, or in other words, sharing a link is allowed. This is a natural requirement due to the non-blocking

property of the Internet. In order to capture the I/O speed constraint in the switch model, we can replace every datacenter with a gadget of two nodes. The first node has exactly the same neighbors and edges that the original node for the datacenter has, plus links from and to the second node. The second node is only connected to the first node, and is the true source and destination for all demands involving this datacenter. By setting capacity on the links between these two nodes, we can enforce I/O limit for the whole datacenter like in switch model. Jahanjou et al. [2] considered the model in which data has to travel along a single specified path. In addition to this model, we also consider the *free path* model which allows data to be split or merged at nodes to utilize the whole graph when transmitting the same piece of data as long as the capacity of each link is respected. This seems much more complicated in practice than a single path transmission, but modern distributed computation frameworks (with software defined network and link aggregation) allow this kind of fine-grained control on network routing and transfer rate, which makes the model realistic.

In Chapter 3, joint work with Mosharaf Chowdhury, Samir Khuller, Manish Purohit, and Jie You [17], we give a tight 2-approximation algorithm based on LP rounding. In Section 5.1.1 we cover future directions on this topic (on both the switch model and the graph model).

1.3 Scheduling with Spot Instance

Cloud computing providers, due to economy of scale, are in a better position in providing cheap computational resources than smaller entities. To cope with periodic peak demand, users (the cloud providers themselves included) reserve more resources than they require, leaving the whole system at low utilization. The Spot instance market, e.g. Amazon EC2 Spot Instances [18], Microsoft Low Priority VM [19], Preemptible VM Instances [20], is an attempt to reallocate these unused resources by providing them at a deep discount comparing to normal ones. Users request spot instances, and will be guaranteed if current load is low and there are instances available. When system demands spike up, spot instances may be interrupted and reclaimed by the system with little or no advanced notification. While most tasks are not compatible with spot instances due to interruptions, certain tasks can take advantage of the reduced price with limited impact. Such tasks usually have a best-effort property and can safely save to and restore from checkpoints. Acting as a cheap computational solution, spot instances can dramatically increase the computation resources available within a budget. One example that would benefit is the training of machine learning models: modern training frameworks can handle interruptions, and more training epochs increase the quality of the model.

We give a model for the general problem of scheduling tasks (especially ML training jobs) on spot instances given a budget. This model captures three major challenges. The first challenge is the unpredictable and stochastic nature of interruptions. It is tempting to use the cheapest instance available that can handle the

computation, but frequent interruptions may render this choice unfavorable. Premature interruptions may cause losses in unsaved progress and incur additional costs for restoration, which means we may end up wasting our budget on the overheads of repeated rescheduling without making much real progress. On the other hand, more reliable instances may have a higher hourly rate or slower speed, rendering it also sub-optimal. A good scheduling strategy would require a better trade-off between low price and uninterrupted computations. The second challenge comes from the rescheduling of jobs. When a job is interrupted, we commonly reschedule it on some other available instance. The new instance, in turn, may also get interrupted. Such recursive behavior can be captured by dynamic programming, provided that the utility of a job is additive in its progress, which is generally not true for ML training jobs (details described in the next challenge). Dynamic programming may also fail to work if the availability of instances may lead to exponential number of different states. This could happen, for example, when an interruption on a particular instance indicates a high demand on it, which suggests we avoid it in later schedules. The last challenge originates from the non-linear relationship between the utility we obtained and the total progress we have on jobs. For most scheduling problems, a job only earns its utility when it is completely finished. ML training jobs, on the other hand, gets a valid model after each epoch, but with degraded performance, or partial utilities. Such a partial utility is not linear in the processing time, since doubling the number of training epochs will not double our utility due to the diminishing returns in training.

We model this problem as follows. We have N jobs that can be scheduled on M

instances of cloud servers. Each job may run at different speed on different instances, among which some are spot instances that may be interrupted prematurely. Once interrupted, a job can be rescheduled to an available instance (possible a subset of all instances depending on what schedule has been carried out) and resume from the last checkpoint. A job can also be kept inactive indefinitely if it is no longer profitable due to diminishing returns. We assume the distributions of interruptions are given, and would like to maximize the expected utility under a given budget. To model the diminishing returns in the total utility, we use submodular functions [1], a tool with wide applications in fields as crowdsourcing [21], information gathering [22], sensor placement [23], influence maximization [24, 25] and exemplar-based clustering [26]. This problem is then mapped to the correlated stochastic knapsack problem with a submodular target function.

In Chapter 4, joint work with Sunav Choudhary, Subrata Mitra, Kanak Mahadik, Samir Khuller[27], we present an algorithm that computes an adaptive policy for this problem which is guaranteed to achieve $(1 - 1/\sqrt{e})/2 \simeq 0.1967$ of the optimal solution. It improves on the $(1-1/\sqrt[4]{(e)})/2 \simeq 0.1106$ approximation ratio from Fukunaga et al. [28]. Furthermore, we remove an assumption in Fukunaga et al. [28], which assumes that possible overflow of the budget is not allowed.

1.4 Outline of the Dissertation

We organize the dissertation as follows.

[1] A function $f : 2^{\mathcal{N}} \to \mathbb{R}$ is *submodular* if for every $A \subseteq B \subseteq \mathcal{N}$ and $e \in \mathcal{N} : f(A \cup \{e\}) - f(A) \geq f(B \cup \{e\}) - f(B)$. An equivalent definition is that for every $A, B \subseteq \mathcal{N} : f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$.

In Chapter 2, we consider the coflow scheduling problem and give improved poly-time approximation algorithms for both zero release time and arbitrary release time. In addition, the algorithm proposed is combinatorial and does not require solving a linear program.

In Chapter 3, we extend the coflow scheduling problem from the switch model to the network model. We give a tight approximation algorithm and a matching proof on hardness of approximation.

In Chapter 4, we consider the problem of scheduling on spot instances. We map it a correlated stochastic knapsack problem that maximizes a submodular function. We improve on the approximation ratio for this problem.

In Chapter 5, we conclude the dissertation.

Chapter 2: Coflow Scheduling in Switch Model

In this chapter, we formally define the coflow scheduling problem and give detailed solutions. In Section 2.1, we describe the related works on coflow scheduling and its relationship with the concurrent open shop problem. In Section 2.3 we introduce some notations and explain how to schedule a single coflow. Section 2.4 covers the high-level ideas of our algorithms. Section 2.5 gives the approximation algorithms with zero release time and arbitrary release time. In Section 2.6, we analyze these algorithms. The LP rounding version of the same algorithms are described in Section 2.7. As a side result, Section 2.8 describes a 3-approximation for concurrent open shop problem. In Section 2.9 and Section 2.10, we give a fix to the algorithm in Qiu et al. [13], and a counter example disproving a key lemma in Luo et al. [7].

2.1 Related Works

The idea of scheduling coflows was first introduced by Chowdhury and Stoica [1] to describe the prevalent communication patterns in data centers. Since then, it has been a hot topic in both the systems [7, 8, 9, 10, 11, 12] and the theory [2, 13, 14, 15, 16] communities.

For the special case when all coflows have zero release time, Qiu et al. [13] established the first polynomial-time constant approximation for this problem. They obtained a deterministic $\frac{64}{3}$ approximation and a randomized $(8 + \frac{16\sqrt{2}}{3})$ approximation algorithm for the problem of minimizing the weighted completion time. For coflow scheduling with arbitrary release times, Qiu et al. [13] claimed a deterministic $\frac{67}{3}$ approximation and a randomized $(9 + \frac{16\sqrt{2}}{3})$ approximation algorithm. However in Section 2.9, we demonstrate a subtle error in their proof that deals with non-zero release times. We show that their techniques in fact only yield a deterministic $\frac{76}{3}$-approximation algorithm for coflow scheduling with release times. Their result still holds for the case with equal release times. Khuller and Purohit [14] obtained a deterministic 12-approximation algorithm for coflow scheduling with arbitrary release times. For the special case when all release times are zero they obtained a deterministic 8-approximation and a randomized $3 + 2\sqrt{2} \approx 5.83$-approximation. Their approach is based on reducing coflow scheduling to the concurrent open shop scheduling problem (more detail in Section 2.1.1).

The current approximation ratio for coflow scheduling are due to Chapter 2, a deterministic 5-approximation algorithm with arbitrary release times, and a 4-approximation without release time. In an independent work, Shafiee and Ghaderi [16] obtained the same approximation ratio. Both works need to solve an linear program. We go further in Chapter 2 and get a more practical primal-dual based algorithm that achieves the same approximation bound for both cases. This result actually transfers to the system community. A system called Sincronia [3] was also developed based on the primal-dual method. It improves upon state-of-the-art

methods and gives practical and near-optimal solutions in real testbeds.

For the online case, Khuller et al. [29] study coflow scheduling in the online setting where the coflows arrive online over time. Using the results of this chapter (Theorem 2), they obtained an exponential time 7-competitive algorithm and a polynomial time 14-competitive algorithm. Since preemption often incurs large overheads, some recent work [12] has tackled the problem of non-preemptive coflow scheduling. Mao et al. [30] consider the non-preemptive coflow scheduling problem with stochastic sizes and give an algorithm with an approximation factor of $(2\log m+1)(1+\sqrt{m}\Delta)(1+m\Delta)(3+\Delta)/2$, where Δ is an upper bound of squared coefficient of variation of processing times. This simplifies to a $(3\log m+\frac{3}{2})$ approximation for non-stochastic cases.

2.1.1 Relationship with Concurrent Open Shop

The coflow scheduling problem generalizes the well-studied concurrent open shop problem [31, 32, 33, 34, 35]. In the concurrent open shop problem, we have a set of m machines and each job j (with weight w_j) is composed of m tasks $\{t_i^j\}_{i=1}^m$, one on each machine. Let p_i^j denote the processing requirement of task t_i^j. A job j is considered completed once all its tasks have completed. A machine can perform at most one unit of processing at a time. The goal is to find a feasible schedule that minimizes the total weighted completion time of jobs. An LP-relaxation yields a 2-approximation algorithm for concurrent open shop scheduling when all release times are zero [31, 32, 33] and a 3-approximation algorithm for arbitrary release

times [32, 33]. The approximation ratio is improved to 2 by Im et al. [36], but with an LP-based method. Mastrolilli et al. [34] show that a simple primal-dual algorithm also yields a 2-approximation for concurrent open shop without release times. We develop a primal-dual algorithm that yields a 3-approximation for concurrent open shop with release times.

The concurrent open shop problem can be viewed as a special case of coflow scheduling when the demand matrices D^j are diagonal for all coflows j [9, 13]. We use our algorithm to get a schedule for coflows and then schedule the jobs in the same order. Since coflow is preemptive by definition, our algorithm gives a preemptive schedule. One might think by reducing the concurrent open shop problem to coflow scheduling we get a preemptive schedule. However for the case when all release times are zero, the schedule is automatically non-preemptive.

At first glance, it appears that coflow scheduling is much harder than concurrent open shop. For instance, while concurrent open shop always admits an optimal permutation schedule, such a property does not hold for coflows [9]. Surprisingly, we show that using a similar LP relaxation as for the concurrent open shop problem, we can design a primal dual algorithm to obtain a permutation of coflows such that sequentially scheduling the coflows after some post-processing in this permutation leads to provably good coflow schedules. Since the coflow scheduling problem generalizes the well-studied concurrent open shop scheduling problem, it is NP-hard to approximate within a factor better than $(2 - \epsilon)$ [37, 38]. For the concurrent open shop scheduling problem, an LP-relaxation yields a 2-approximation algorithm when all release times are zero [31, 32, 33] and a 3-approximation algorithm for arbitrary

release times [32, 33]. Mastrolilli et al. [34] showed a 2-approximation for concurrent open shop without release times. By exploiting a connection with the well-studied concurrent open shop scheduling problem, Luo et al. [7] claim a 2-approximation algorithm for coflow scheduling when all the release times are zero. Unfortunately, as we show in Section 2.10, their proof is flawed and the result does not hold.

2.2 Our Contributions

The main algorithmic contribution of this chapter is a deterministic, primal-dual algorithm for the offline coflow scheduling problem with improved approximation guarantees.

Theorem 1. *There exists a deterministic, combinatorial, polynomial time 5-approximation algorithm for coflow scheduling with release times.*

Theorem 2. *There exists a deterministic, combinatorial, polynomial time 4-approximation algorithm for coflow scheduling without release times.*

Our results significantly improve upon the approximation algorithms developed by Khuller and Purohit [14] whose techniques yield a 12-approximation algorithm for the case with release time, and an 8-approximation algorithm without release time. In addition, our algorithm is completely combinatorial and does not require solving a linear program. An LP-based version is also provided together with its proof, to help show the intuition behind the primal-dual one.

We also extend the primal dual algorithm by Mastrolilli et al. [34] to give a 3-approximation algorithm for the concurrent open shop problem when the jobs have

arbitrary release times. Leung et al. [33] have a LP based algorithm which gives a 3-approximation as well, but our approach is the first combinatorial algorithm which achieves this bound.

Theorem 3. *There exists a deterministic, combinatorial, polynomial time 3-approximation algorithm for concurrent open shop scheduling with release times.*

2.3 Preliminaries

We first introduce some notation to facilitate the following discussion. For every coflow j and input port i, we define the load $L_{i,j} = \sum_{o=1}^{m} d_{io}^{j}$ to be the total amount of data that coflow j needs to transmit through input port i. Similarly, we define $L_{o,j} = \sum_{i=1}^{m} d_{io}^{j}$ for every coflow j and output port o. Equivalently, a coflow j can be represented by a weighted, bipartite graph $G_j = (I, O, E_j)$ where the set of input ports (I) and the set of output ports (O) form the two sides of the bipartition and an edge $e = (i, o)$ with weight $w_{G_j}(e) = d_{io}^{j}$ represents that the coflow j requires d_{io}^{j} units of data to be transferred from input port i to output port o. We will abuse notations slightly and refer to a coflow j by the corresponding bipartite graph G_j when there is no confusion.

Representing a coflow as a bipartite graph simplifies some of the notation that we have seen previously. For instance, for any coflow j, the load of j on port i is simply the weighted degree of vertex i in graph G_j, i.e., if $\mathbb{N}_{G_j}(i)$ denotes the set of

neighbors of node i in the graph G_j.

$$L_{i,j} = \mathsf{deg}_{G_j}(i) = \sum_{o \in \mathbb{N}_{G_j}(i)} w_{G_j}(i, o) \tag{2.1}$$

For any graph G_j, let $\Delta(G_j) = \max_{s \in I \cup O} \mathsf{deg}_{G_j}(s) = \max\{\max_i L_{i,j}, \max_o L_{o,j}\}$ denote the maximum degree of any node in the graph, i.e., the load on the most heavily loaded port of coflow j.

In our algorithm, we consider coflows obtained as the union of two or more coflows. Given two weighted bipartite graphs $G_j = (I, O, E_j)$ and $G_k = (I, O, E_k)$, we define the cumulative graph $G_j \cup G_k = (I, O, E_j \cup E_k)$ to be a weighted bipartite graph such that $w_{G_j \cup G_k}(e) = w_{G_j}(e) + w_{G_k}(e)$. We extend this notation to the union of multiple graphs in an obvious manner.

2.3.1 Scheduling a Single Coflow

Before we present our algorithm for the general coflow scheduling problem, it is instructive to consider the problem of feasibly scheduling a *single coflow* subject to the matching constraints. Given a coflow G_j, the maximum degree of any vertex in the graph $\Delta(G_j) = \max_v \mathsf{deg}_{G_j}(v)$ is an obvious lower bound on the amount of time required to feasibly schedule coflow G_j. In fact, the following lemma by Qiu et al. [13] shows that this bound is always achievable for any coflow. The proof follows by repeated applications of Hall's Theorem on the existence of perfect matchings in bipartite graphs.

Lemma 1. *[13] There exists a polynomial time algorithm that schedules a single coflow G_j in $\Delta(G_j)$ time steps.*

Lemma 1 also implicitly provides a way to decompose a bipartite graph G into two graphs G_1 and G_2 such that $\Delta(G) = \Delta(G_1) + \Delta(G_2)$. Given a time interval $(t_s, t_e]$, the following corollary uses such a decomposition to obtain a feasible coflow schedule for the given time interval by partially scheduling a coflow if necessary.

Corollary 1. *Given a sequence of coflows $G_1, G_2, \ldots, G_n$, a start time t_s, and an end time t_e such that $t_s + \sum_{k=1}^{j-1} \Delta(G_k) \leq t_e < t_s + \sum_{k=1}^{j} \Delta(G_k)$, there exists a polynomial time algorithm that finds a feasible coflow schedule for the time interval $(t_s, t_e]$ such that -*

- *coflows $G_1, G_2, \ldots, G_{j-1}$ are completely scheduled.*
- *coflow G_j is partially scheduled so that $\Delta(\tilde{G}_j) = t_s + \sum_{k=1}^{j} \Delta(G_k) - t_e$ where $\tilde{G}_j$ denotes the subset of coflow j that has not yet been scheduled.*
- *coflows $G_{j+1}, \ldots, G_n$ are not scheduled.*

Proof. By scheduling coflows $G_1, G_2, \ldots, G_{j-1}$ sequentially using Lemma 1, we can completely schedule these coflows by time $t_s + \sum_{k=1}^{j-1} \Delta(G_k) \leq t_e$. Similarly using Lemma 1, we find a schedule S for coflow G_j that requires $\Delta(G_j)$ time steps. We schedule only the first $t_e - (t_s + \sum_{k=1}^{j-1} \Delta(G_k))$ matchings from S after all the previous coflows have been completed. This partial scheduling of coflow G_j ends at time t_e as desired. Let $\tilde{G}_j \subset G_j$ denote the partial coflow that has not yet been scheduled. Inspecting schedule S, we observe that S schedules the partial

coflow $\tilde{G}_j$ from time steps $t_e - (t_s + \sum_{k=1}^{j-1} \Delta(G_k))$ to $\Delta(G_j)$. Hence, we must have $\Delta(\tilde{G}_j) \leq t_s + \sum_{k=1}^{j} \Delta(G_k) - t_e$. □

2.3.2 Linear Programming Relaxation

By exploiting the connection with concurrent open-shop scheduling, we adapt the LP relaxation used for the concurrent open-shop problem [32, 33] to formulate the following linear program as a relaxation of the coflow scheduling problem. We introduce a variable C_j for every coflow j to denote its completion time. Let $J = \{1, 2, \ldots, n\}$ denote the set of all coflows and $M = I \cup O$ denote the set of all the ports. Figure 2.1 shows our LP relaxation.

$$\min \sum_{j \in J} w_j C_j$$

$$\text{subject to,} \quad C_j \geq r_j + L_{i,j} \qquad \forall j \in J, \forall i \in M \tag{2.2}$$

$$\sum_{j \in S} L_{i,j} C_j \geq \frac{1}{2} \left(\sum_{j \in S} L_{i,j}^2 + \left(\sum_{j \in S} L_{i,j} \right)^2 \right) \qquad \forall i \in M, \forall S \subseteq J \tag{2.3}$$

Figure 2.1: LP_1 for Coflow Scheduling

The first set of constraints (2.2) ensure that the completion time of any job j is at least its release time r_j plus the load of coflow j on any port i. The second set of constraints (2.3) are standard in scheduling literature (e.g. [39]) and are used to effectively lower bound the completion time variables. For simplicity, we define $f_i(S)$ for any subset $S \subseteq J$ and each port i as follow

$$f_i(S) = \frac{\sum_{j \in S} L_{i,j}^2 + (\sum_{j \in S} L_{i,j})^2}{2} \tag{2.4}$$

2.4 High Level Ideas

We use the LP above in Fig 2.1 and its dual to develop a combinatorial algorithm (Algorithm 1) in Section 2.5.1 to obtain a good permutation of the coflows. This primal dual algorithm is inspired by Davis et al. [40] and Mastrolilli et al. [34]. As we show in Lemma 5, once the coflows are permuted as per this algorithm, we can bound the completion time of a coflow j in an optimal schedule in terms of $\Delta(\bigcup_{k\leq j} G_k)$, the maximum degree of the union of the first j coflows in the permutation.

A naïve approach now would be to schedule each coflow independently and sequentially using Lemma 1 in this permutation. Since all coflows $k \leq j$ would need to be scheduled before starting to schedule j, the completion time of coflow j under such a scheme would be $\sum_{k\leq j} \Delta(G_k)$. Unfortunately, for arbitrary coflows we can have $\sum_{k\leq j} \Delta(G_k) \gg \Delta(\bigcup_{k\leq j} G_k)$. For instance, Fig 2.2 shows three coflows such that $\Delta(G_1) + \Delta(G_2) + \Delta(G_3) = 300 > \Delta(G_1 \cup G_2 \cup G_3) = 101$.

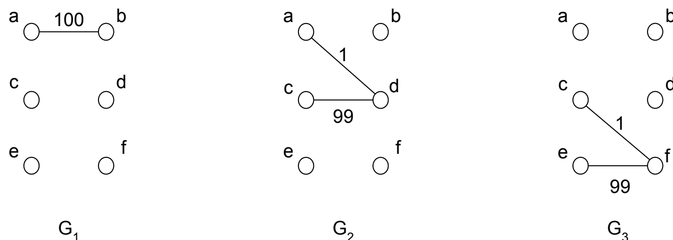

Figure 2.2: Example that illustrates sequentially scheduling coflows independently can lead to bad schedules.

One key insight is that sequentially scheduling coflows one after another may waste resources, such as in Fig 2.2. Since the amount of time required to completely

schedule a single coflow k only depends on the maximum degree of the graph G_k, if we augment graph G_k by adding edges such that its maximum degree does not increase, the augmented coflow can still be scheduled in the same time interval. This observation leads to the natural idea of "shifting" edges from a coflow j later in the permutation to an earlier coflow k ($k < j$), so long as the release time of j is still respected, as such a shift does not delay coflow k but may significantly reduce the requirements of coflow j. Consider for instance the coflows in Figure 2.2 when all release times are zero; shifting the edge (c, d) from graph G_2 to G_1 and the edges (e, f) and (c, f) from G_3 to G_1 leaves $\Delta(G_1)$ unchanged but drastically reduces $\Delta(G_2)$ and $\Delta(G_3)$. Let's call the coflows after shifting edges G'_1, G'_2, G'_3. After moving edges, $\Delta(G'_1) = 100$, $\Delta(G'_2) = 1$ and $\Delta(G'_3) = 0$. If we schedule G'_1, G'_2, G'_3 sequentially, completion time of G_1 (C_1) will be $\Delta(G'_1) = 100$, $C_2 = \Delta(G'_1) + \Delta(G'_2) = 101$ and $C_3 = \Delta(G'_1) = 100$. Before shifting edges, on the country, completion times were $C_1 = 100, C_2 = 200, C_3 = 300$. Thus shifting edges reduces completion times.

In Algorithm 3 in Section 3.4.1, we formalize this notion of shifting edges and prove that after all such edges have been shifted, sequentially scheduling the augmented coflows leads to a provably good coflow schedule.

In Section 2.7 we present an alternative approach using LP Rounding for finding a good permutation of coflows. Then we schedule the coflows using Algorithm 3 and give proofs for the non-combinatorial version of Theorem 1 and Theorem 2.

2.5 Approximation Algorithm for Coflow Scheduling with Release Times

In this section we present a combinatorial 5-approximation algorithm for minimizing the weighted sum of completion times of a set of coflows with release times. Our algorithm consists of two stages. In the first stage, we design a primal-dual algorithm to find a good permutation of the coflows. In the second stage, we show that scheduling the coflows sequentially in this ordering after some postprocessing steps yields a provably good coflow schedule.

2.5.1 Finding a Permutation of Coflows Using a Primal Dual Algorithm

Although our algorithm does not require solving a linear program, we use the linear program in Figure 2.1 and its dual (Figure 2.3) in the design and analysis of the algorithm.

$$\begin{aligned}
&\max \sum_{j\in J}\sum_{i\in M}\alpha_{i,j}(r_j+L_{i,j})+\sum_{i\in M}\sum_{S\subseteq J}\beta_{i,S}f_i(S) && \\
\text{subject to,}\quad &\sum_{i\in M}\alpha_{i,j}+\sum_{i\in M}\sum_{S/j\in S}L_{i,j}\beta_{i,S}\le w_j && \forall j\in J\\
&\alpha_{i,j}\ge 0 && \forall j\in J, i\in M\\
&\beta_{i,S}\ge 0 && \forall i\in M,\forall S\subseteq J
\end{aligned}$$

Figure 2.3: Dual of LP_1

Our algorithm works as follows. We build up a permutation of the coflows

in reverse order iteratively. Let κ be a constant that we specify later. Let J be the set of unscheduled jobs, initially $J = \{1, 2, \cdots, n\}$. In any iteration, let j be the unscheduled job with the latest release time, let μ be the port with the highest overall load and let L_μ be the load on port μ. Now if $r_j > \kappa L_\mu$, we raise the dual variable $\alpha_{\mu,j}$ until the corresponding dual constraint is tight and place coflow j at the last in the permutation. But if $r_j \leq \kappa L_\mu$, we raise the dual variable $\beta_{\mu,J}$ until the dual constraint for some job j' becomes tight and place coflow j' at the last in the permutation. Algorithm 1 gives the formal description of the complete algorithm.

Algorithm 1: Permuting Coflows

1 J is the set of unscheduled jobs and initially $J = \{1, 2, \cdots, n\}$;
2 Initialize $\alpha_{i,j} = 0$ for all $i \in M, j \in J$ and $\beta_{i,S} = 0$ for all $i \in M, S \subseteq J$;
3 $L_i = \sum_{j \in J} L_{ij}, \forall i \in M$; `// load of port` i
4 **for** $k = n, n-1, \cdots, 1$ **do**
5 $\quad \mu(k) = \arg\max_{i \in M} L_i$; `// determine the port with highest load`
6 $\quad j = \arg\max_{\ell \in J} r_\ell$; `// determine job that released last`
7 $\quad$ **if** $r_j > \kappa \cdot L_{\mu(k)}$ **then**
8 $\quad\quad \alpha_{\mu(k),j} = (w_j - \sum_{i \in M} \sum_{S \ni j} L_{i,j} \beta_{i,S})$;
9 $\quad\quad \sigma(k) \leftarrow j$;
10 $\quad$ **end**
11 $\quad$ **else if** $r_{\sigma(k)} \leq \kappa \cdot L_{\mu(k)}$ **then**
12 $\quad\quad j' = \arg\min_{j \in J} \left(\frac{w_j - \sum_{i \in M} \sum_{S \ni j} L_{i,j} \beta_{i,S}}{L_{\mu(k),j}} \right)$;
13 $\quad\quad \beta_{\mu(k),J} = \left(\frac{w_{j'} - \sum_{i \in M} \sum_{S \ni j'} L_{i,j'} \beta_{i,S}}{L_{\mu(k),j'}} \right)$;
14 $\quad\quad \sigma(k) \leftarrow j'$;
15 $\quad$ **end**
16 $\quad J \leftarrow J \setminus \sigma(k)$;
17 $\quad L_i \leftarrow L_i - L_{i,\sigma(k)}, \forall i \in M$;
18 **end**
19 Output permutation $\sigma(1), \sigma(2), \cdots, \sigma(n)$;

2.5.2 Scheduling Coflows According to a Permutation

We assume without loss of generality that the coflows are ordered based on the permutation given by Algorithm 1, i.e. $\sigma(j) = j$.

As we discussed in Section 2.4, naïvely scheduling the coflows sequentially in this order may not be a good idea. However, by appropriately moving edges from a coflow j to an earlier coflow k ($k < j$), we can get a provably good schedule. The crux of our algorithm lies in the subroutine MoveEdgesBack defined in Algorithm 2.

Algorithm 2: The MoveEdgesBack subroutine.

```
1 Function MoveEdgesBack(G_k, G_j)
2     for e = (u, v) ∈ G_j do
3         δ = min(Δ(G_k) − deg_{G_k}(u), Δ(G_k) − deg_{G_k}(v), w_{G_j}(e));
4         w_{G_j}(e) = w_{G_j}(e) − δ;
5         w_{G_k}(e) = w_{G_k}(e) + δ;
6     end
7     return G_k, G_j;
```

Given two bipartite graphs G_k and G_j ($k < j$), MoveEdgesBack greedily moves weighted edges from graph G_j to G_k so long as the maximum degree of graph G_k does not increase. The key idea behind this subroutine is that since the coflow k requires $\Delta(G_k)$ time units to be scheduled feasibly, the edges moved back can now also be scheduled in those $\Delta(G_k)$ time units for "free".

If all coflows have zero release times, then we can safely move edges of a coflow G_j to any G_k such that $k < j$. However, with the presence of arbitrary release times, we need to ensure that edges of coflow G_j do not violate their release time, i.e. they are scheduled only after they are released. Algorithm 3 describes the pseudo-code for coflow scheduling with arbitrary release times. Here q denote the number of distinct

values taken by the release times of the n coflows. Further, let $t_1 < t_2 < \ldots < t_q$ be the ordered set of the release times. For simplicity, we define $t_{q+1} = T$ as a sufficiently large time horizon.

At any time step t_i, let $G'_j \subseteq G_j$ denote the subgraph of coflow j that has not been scheduled yet. We consider every ordered pair of coflows $k < j$ such that both coflows are released by time t and MoveEdgesBack from graph G'_j to graph G'_k. Finally, we schedule the coflows sequentially in the order using Corollary 1 until all coflows are scheduled completely or we reach time t_{i+1} when a new set of coflows gets released and the process repeats.

Algorithm 3: Coflow Scheduling

```
1  q ← number of distinct release times;   t_{q+1} ← T;
2  t_1, t_2, ..., t_q ← distinct release time in increasing order ;
3  for  i = 1, 2, ..., q do
4  |  // Each loop finds a schedule for time interval (t_i, t_{i+1}]
5  |  for  j = 1, 2, ..., n do
6  |  |  G'_j ← unscheduled part of G_j;
7  |  end
8  |  for  k = 1, 2, ..., n − 1 do
9  |  |  if  r_k ≤ t_i then
10 |  |  |  for  j = k + 1, ..., n do
11 |  |  |  |  if  r_j ≤ t_i then  G'_k, G'_j ← MoveEdgesBack(G'_k, G'_j) ;
12 |  |  |  end
13 |  |  end
14 |  end
15 |  Schedule (G'_1, G'_2, ..., G'_n) in (t_i, t_{i+1}] using Corollary 1;
16 end
```

2.6 Analysis

We first analyze Algorithm 3 and upper bound the completion time of a coflow j in terms of the maximum degree of the cumulative graph obtained by combining the first j coflows in the given permutation. For simplicity, we first state the proof when all release times are zero, then proceed to the case with arbitrary release time

2.6.1 Coflows with Zero Release Times

For ease of presentation, we first analyze the special case when all coflows are released at time zero. In this case, we have $q = 1$ in Algorithm 3, so the outer *for* loop is only executed once. The following lemma shows that after the MoveEdgesBack subroutine has been executed on every ordered pair of coflows, for any coflow j, the sum of maximum degrees of graphs G'_k ($k \leq j$) is at most twice the maximum degree of the cumulative graph obtained by combining the first j coflows.

Lemma 2. *For all $j \in \{1, 2, \ldots n\}$, $\sum_{k \leq j} \Delta(G'_k) \leq 2\Delta(\bigcup_{k \leq j} G_k)$.*

Proof. Since the graphs G'_k keep changing during the course of the algorithm, for the sake of analysis, let $G_{k|j}$ where $k < j$ be the state of the graph G'_k immediately after we have transferred all possible edges from G'_j to G'_k. Let $G_{j|j}$ denote the graph G'_j after all possible edges have been moved to G'_{j-1}. Since we move edges back to a graph G'_k only if it does not increase the maximum degree, we have the following:

$$\Delta(G'_k) = \Delta(G_{k|j}) \text{ for all } k \leq j. \tag{2.5}$$

For any $j \in \{1, 2, \ldots, n\}$, consider the set S of graphs $G_{1|j}, G_{2|j}, \ldots G_{j|j}$. Let u be a vertex of maximum degree in $G_{j|j}$, i.e. $deg_{G_{j|j}}(u) = \Delta(G_{j|j})$ and consider any edge $e = (u, v)$ incident on u in $G_{j|j}$. Since edge (u, v) was not moved to any of the graphs $G_{k|j}$ for $k < j$, we must have that either u or v had maximum degree in $G_{k|j}$. Let $S_u = \{G_{k|j} \mid deg_{G_{k|j}}(u) = \Delta(G_{k|j})\}$ and $S_v = \{G_{k|j} \mid deg_{G_{k|j}}(v) = \Delta(G_{k|j})\}$ denote the subsets of the graph where vertex u or v has the maximum degree respectively.

Now, let $\hat{G}_j = \bigcup_{k=1}^{j} G_{k|j}$ be the union of the graphs $G_{k|j}$. Since $\hat{G}_j$ contains all edges from the graphs $G_1, \ldots, G_j$ and no edges from graphs G_l for $l > j$, $\hat{G}_j$ is identical to the cumulative graph of the first j coflows. In particular, we have the following:

$$\Delta(\hat{G}_j) = \Delta(\bigcup_{k \leq j} G_k). \tag{2.6}$$

Let us now consider the maximum degree of the graph $\hat{G}_j$.

$$\Delta(\hat{G}_j) \geq \max\left\{\mathsf{deg}_{\hat{G}_j}(u), deg_{\hat{G}_j}(v)\right\} \tag{2.7}$$

$$\geq \max\left\{\sum_{G \in S_u} \mathsf{deg}_G(u), \sum_{G \in S_v} \mathsf{deg}_G(v)\right\} \tag{2.8}$$

$$= \max\left\{\sum_{G \in S_u} \Delta(G), \sum_{G \in S_v} \Delta(G)\right\} \tag{2.9}$$

From Equation (2.5), we have the following:

$$\sum_{k \leq j} \Delta(G'_k) = \sum_{k \leq j} \Delta(G_{k|j}) = \sum_{G \in S} \Delta(G). \tag{2.10}$$

However, since $S_u \cup S_v = S$ as either u or v has maximum degree in every graph in

S, we get the following.

$$\sum_{k\leq j}\Delta(G'_k)\leq 2\max\left\{\sum_{G\in S_u}\Delta(G),\sum_{G\in S_v}\Delta(G)\right\}\leq 2\Delta(\hat{G}_j)=2\Delta(\bigcup_{k\leq j}G_k)$$

where the last equality follows from Equation (2.6). □

Lemma 3. *Consider any coflow j and let $C_j(alg)$ denote the completion time of coflow j when scheduled as per Algorithm 3. Then $C_j(alg)\leq 2\Delta(\bigcup_{k\leq j}G_k)$.*

Proof. Let $G'_1,\ldots,G'_n$ denote the coflows after all the edges have been moved backward. According to Lemma 1 each coflow G'_k could be finished at time $\Delta(G'_k)$, thus when the coflows are scheduled sequentially, we get the following.

$$C_j(alg)=\sum_{k\leq j}\Delta(G'_k)\leq 2\Delta(\bigcup_{k\leq j}G_k)$$

where the last inequality follows from Lemma 2. □

2.6.2 Coflows with Arbitrary Release Times

When the coflows have arbitrary release times, we can bound the completion time of each coflow j in terms of the maximum degree of the cumulative graph obtained by combining the first j coflows and the largest release time of all the jobs before j in the permutation.

Lemma 4. *For any coflow j, let $C_j(alg)$ denote the completion time of coflow j when scheduled as per Algorithm 3. Then $C_j(alg)\leq \max_{k\leq j} r_k+2\Delta(\bigcup_{k\leq j}G_k)$*

Proof. Consider any coflow j. Let $t_i = \max_{l \leq j} r_l$ denote the earliest time when all coflows in the set $\{1, 2, \ldots, j\}$ have been released. In Algorithm 3, consider the ith iteration of the for loop. Let $G_{k,i}$ denote the graph corresponding to coflow k in iteration i before edges have been moved back, i.e., $G_{k,i}$ denotes the state of coflow k in iteration i after line 7. Since some edges from coflow k may have already been scheduled in earlier iterations, we have $G_{k,i} \subseteq G_k$. Let $G'_{k,i}$ denote the graph corresponding to coflow k after the MoveEdgesBack subroutines have been executed, i.e. at line 14. We now claim that

$$C_j(alg) \leq t_i + \sum_{k \leq j} \Delta(G'_{k,i}) \tag{2.11}$$

If $t_{i+1} \geq t_i + \sum_{k \leq j} \Delta(G'_{k,i})$, Corollary 1 guarantees that coflows $1 \leq k \leq j$ will be completely scheduled sequentially in this iteration. Completion time of coflow j is thus $t_i + \sum_{k \leq j} \Delta(G'_{k,i})$ as desired.

On the other hand, if $t_{i+1} < t_i + \sum_{k \leq j} \Delta(G'_{k,i})$, let p denote the first coflow such that $t_{i+1} < t_i + \sum_{k \leq p} \Delta(G'_{k,i})$. Corollary 1 now finds feasible schedules for time slots t_i to t_{i+1} such that all coflows $k \leq p - 1$ are completely scheduled and coflow p is partially scheduled so that we have the following:

$$\Delta(G'_{p,i+1}) = \Delta(G_{p,i+1}) = t_i + \sum_{k \leq p} \Delta(G'_{k,i}) - t_{i+1} \tag{2.12}$$

$$\Delta(G'_{k,i+1}) = \Delta(G_{k,i+1}) = 0, \forall k \leq p - 1. \tag{2.13}$$

Also, since all the coflows $1 \leq k \leq j$ had already been released at time t_i, any

new coflows that get released do not affect the movement of edges from graphs corresponding to coflows $1 \leq k \leq j$. Hence, we have:

$$\Delta(G'_{k,i+1}) = \Delta(G'_{k,i}), \forall p < k \leq j \tag{2.14}$$

From equations (2.12) - (2.14), we get:

$$t_{i+1} + \sum_{k \leq j} \Delta(G'_{k,i+1}) = t_i + \sum_{k \leq j} \Delta(G'_{k,i}). \tag{2.15}$$

Proceeding this way inductively, we obtain:

$$t_{i+x} + \sum_{k \leq j} \Delta(G'_{k,i+x}) = t_i + \sum_{k \leq j} \Delta(G'_{k,i}). \tag{2.16}$$

where $i + x$ is the last iteration such that $t_{i+x} < t_i + \sum_{k \leq j} \Delta(G'_{k,i})$. By Corollary 1 at the end of iteration $i + x$, coflow j is completely scheduled at time $t_{i+x} + \sum_{k \leq j} \Delta(G'_{k,i+x}) = t_i + \sum_{k \leq j} \Delta(G'_{k,i})$ as desired, thus completing the proof of the claim.

We can now bound $C_j(alg)$ as follows.

$$C_j(alg) \leq t_i + \sum_{k \leq j} \Delta(G'_{k,i}) \leq t_i + 2\Delta\left(\bigcup_{k \leq j} G_{k,i}\right) \leq t_i + 2\Delta\left(\bigcup_{k \leq j} G_k\right). \tag{2.17}$$

where the second inequality follows from Lemma 2. □

2.6.3 Analyzing the Primal-Dual Algorithm

We are now in a position to analyze Algorithm 1. Recall that we assume that the jobs are sorted as per the permutation obtained by Algorithm 1, i.e., $\sigma(k) = k, \forall k \in [n]$. We first give a lemma, which will be proved in Section .

Lemma 5. *If there is an algorithm that generates a feasible coflow schedule such that for any coflow j, $C_j(alg) \leq a \max_{k \leq j} r_k + b\Delta(\bigcup_{k \leq j} G_k)$ for some constants a and b, then the total cost of the schedule is bounded as follows.*

$$\sum_j w_j C_j(alg) \leq (a + \frac{b}{\kappa}) \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + 2(a\kappa + b) \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S).$$

Theorem 1. *There exists a deterministic, combinatorial, polynomial time 5-approximation algorithm for coflow scheduling with release times.*

Proof. For scheduling coflows with arbitrary release times, Lemmas 4 and 5 (with $a = 1$ and $b = 2$) together imply that:

$$\sum_j w_j C_j(alg) \leq \left(1 + \frac{2}{\kappa}\right) \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + 2(\kappa + 2) \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S).$$

To minimize the approximation ratio, we substitute $\kappa = \frac{1}{2}$ and obtain:

$$\sum_j w_j C_j(alg) \leq 5 \left(\sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S) \right) \leq 5 \cdot OPT,$$

where the last inequality follows from weak duality as α and β constitute a feasible

dual solution. □

Theorem 2. *There exists a deterministic, combinatorial, polynomial time 4-approximation algorithm for coflow scheduling without release times.*

Proof. Lemmas 4 and 5 (with $a = 0$ and $b = 2$) together imply that:

$$\sum_j w_j C_j(alg) \leq \frac{2}{\kappa} \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + 4 \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S).$$

To minimize the approximation ratio, we substitute $\kappa = \frac{1}{2}$ and obtain:

$$\sum_j w_j C_j(alg) \leq 4 \left(\sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S) \right) \leq 4 \cdot OPT,$$

where the last inequality follows from weak duality as α and β constitute a feasible dual solution. □

2.6.4 Primal Dual Analysis

We devote this section to prove Lemma 5.

Recall that we assume that the jobs are sorted as per the permutation obtained by Algorithm 1, i.e., $\sigma(k) = k, \forall k \in [n]$.

Let S_j be the set of jobs $\{1, \cdots, j\}$. Let $\beta_{i,j} = \beta_{i,S_j}$ and $L_i(S_j) = \sum_{k \leq j} L_{i,k}$. Also let $\mu(j)$ be the port with highest load in S_j, therefore $L_{\mu(j)}(S_j) = \sum_{k \leq j} L_{\mu(j),k} = \Delta(\bigcup_{k \leq j} G_k)$. We will first state a few observations regarding the primal-dual algorithm.

Observation 1. *The following statements hold.*

(a) Every nonzero $\beta_{i,S}$ *can be written as* $\beta_{\mu(j),j}$ *for some job* j.

(b) For every set S_j *that has a nonzero* $\beta_{\mu(j),j}$ *variable, if* $k \leq j$ *then* $r_k \leq \kappa \cdot L_{\mu(j)}(S_j)$.

(c) For every job j *that has a nonzero* $\alpha_{\mu(j),j}$, $r_j > \kappa \cdot L_{\mu(j)}(S_j)$.

(d) For every job j *that has a nonzero* $\alpha_{\mu(j),j}$, *if* $k \leq j$ *then* $r_k \leq r_j$.

The correctness of Observation 1 can be directly obtained from Algorithm 1.

Lemma 6. *For every job* j, $\sum_{i\in M} \alpha_{i,j} + \sum_{i\in M} \sum_{k\geq j} L_{i,j}\beta_{i,k} = w_j$.

Proof. A job j is added to the permutation in Algorithm 1 only if the constraint $\sum_{i\in M} \alpha_{i,j} + \sum_{i\in M} \sum_{S/j\in S} L_{i,j}\beta_{i,S} \leq w_j$ gets tight for this job, thus:

$$\sum_{i\in M} \alpha_{i,j} + \sum_{i\in M} \sum_{S/j\in S} L_{i,j}\beta_{i,S} = w_j$$

$$\sum_{i\in M} \alpha_{i,j} + \sum_{i\in M} \sum_{k\geq j} L_{i,j}\beta_{i,k} = w_j.$$

□

Observation 2. *For any* $i \in M$ *and* $S \subseteq J$, *we have that* $(\sum_{j\in S} L_{i,j})^2 \leq 2f_i(S)$.

Lemma 5. *If there is an algorithm that generates a feasible coflow schedule such that for any coflow* j, $C_j(alg) \leq a \max_{k\leq j} r_k + b\Delta(\bigcup_{k\leq j} G_k)$ *for some constants* a

and b*, then the total cost of the schedule is bounded as follows.*

$$\sum_j w_j C_j(alg) \leq (a + \frac{b}{\kappa}) \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + 2(a\kappa + b) \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S).$$

Proof. In the following we denote $C_j(alg)$ as C_j for ease of notation. By applying Lemma 6:

$$\sum_{j=1}^{n} w_j \cdot C_j = \sum_{j=1}^{n} \left(\sum_{i \in M} \alpha_{i,j} + \sum_{i \in M} \sum_{k \geq j} L_{i,j} \beta_{i,k} \right) \cdot C_j$$
$$= \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} \cdot C_j + \sum_{j=1}^{n} \sum_{i \in M} \sum_{k \geq j} L_{i,j} \beta_{i,k} \cdot C_j.$$

First let's bound $\sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} \cdot C_j$. Since $\Delta(\bigcup_{k \leq j} G_k) = L_{\mu(j)}(S_j)$:, by applying Observation 1 parts (c), (d), we get:

$$\sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} \cdot C_j$$
$$\leq \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} \left\{ a \cdot \max_{\ell \leq j} r_\ell + b \cdot L_{\mu(j)}(S_j) \right\}$$
$$\leq \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} \left(a \cdot r_j + b \cdot \frac{r_j}{\kappa} \right)$$
$$\leq \left(a + \frac{b}{\kappa} \right) \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j.$$

Now we bound $\sum_{j=1}^{n} \sum_{i \in M} \sum_{k \geq j} L_{i,j} \beta_{i,k} C_j$:

$$\sum_{j=1}^{n} \sum_{i \in M} \sum_{k \geq j} L_{i,j} \beta_{i,k} C_j$$

$$\leq \sum_{j=1}^{n} \sum_{i \in M} \sum_{k \geq j} L_{i,j} \beta_{i,k} \cdot \left\{ a \cdot max_{\ell \leq j} r_\ell + b \cdot L_{\mu(j)}(S_j) \right\}$$

$$\leq \sum_{j=1}^{n} \sum_{i \in M} \sum_{k \geq j} L_{i,j} \beta_{i,k} \cdot \left\{ a \cdot max_{\ell \leq k} r_\ell + b \cdot L_{\mu(j)}(S_j) \right\}.$$

By applying Observation 1 part (b):

$$\leq \sum_{j=1}^{n} \sum_{i \in M} \sum_{k \geq j} L_{i,j} \beta_{i,k} \cdot \left\{ a\kappa \cdot L_{\mu(k)}(S_k) + b \cdot L_{\mu(j)}(S_j) \right\}$$

$$\leq (a\kappa + b) \sum_{k=1}^{n} \sum_{i \in M} \sum_{j \leq k} L_{i,j} \beta_{i,k} \cdot L_{\mu(k)}(S_k)$$

$$\leq (a\kappa + b) \sum_{k=1}^{n} \sum_{i \in M} \beta_{i,k} \sum_{j \leq k} L_{i,j} \cdot L_{\mu(k)}(S_k)$$

$$= (a\kappa + b) \sum_{k=1}^{n} \sum_{i \in M} \beta_{i,k} \left(L_i(S_k) \right) \cdot L_{\mu(k)}(S_k)$$

$$\leq (a\kappa + b) \sum_{i \in M} \sum_{k=1}^{n} \beta_{i,k} \left(L_{\mu(k)}(S_k) \right)^2.$$

By sequentially applying Observation 2 and Observation 1 part (a), this is upper bounded by

$$2(a\kappa + b) \sum_{i \in M} \sum_{k=1}^{n} \beta_{i,k} f_{\mu(k)}(S_k)$$

$$= 2(a\kappa + b) \sum_{k=1}^{n} \beta_{\mu(k),k} f_{\mu(k)}(S_k)$$

$$\leq 2(a\kappa + b) \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S).$$

Therefore,

$$\sum_{j\in J} w_j C_j \leq \left(a + \frac{b}{\kappa}\right) \sum_{j=1}^{n} \sum_{i\in M} \alpha_{i,\sigma(j)} r_j + 2(a\kappa + b) \sum_{i\in M} \sum_{S\subseteq J} \beta_{i,S} f_i(S).$$

□

2.7 An Alternative Approach Using LP Rounding

This alternative approach also consists of two stages. First, we find a good permutation of coflows and after that we schedule the coflows sequentially in this ordering using Algorithm 3.

Let $\overline{C_j}$ denote the completion time of job j in an optimal $\mathbf{LP_1}$ solution. We assume without loss of generality that the coflows are ordered so that the following holds.

$$\overline{C_1} \leq \overline{C_2} \leq \ldots \leq \overline{C_n} \tag{2.18}$$

We can use the LP-constraints to provide a lower bound on $\overline{C_j}$ in terms of the maximum degree of the cumulative graph obtained by combining the first j coflows. In particular, the following lemma follows from the constraints of $\mathbf{LP_1}$.

Lemma 7. *For each coflow $j = 1, 2, \ldots, n$, the following inequality holds.*

$$\overline{C_j} \geq \frac{1}{2} \max_i \left\{ \sum_{k=1}^{j} L_{i,k} \right\} = \frac{1}{2} \Delta(\bigcup_{k\leq j} G_k)$$

Proof. Let $S = \{1, 2, \ldots, j\}$. The LP constraint (2.3) implies that

$$\max_i \left\{ \sum_{k=1}^{j} L_{i,k} \cdot \overline{C_k} \right\} \geq \max_i f_i(S) \geq \max_i \left\{ \frac{(\sum_{k=1}^{j} L_{i,k})^2}{2} \right\}$$

Since $\overline{C_k} \leq \overline{C_j}$, for each $k = 1, 2, \ldots j$ we have

$$\overline{C_j} \cdot \max_i \{ \sum_{k=1}^{j} L_{i,k} \} = \max_i \left\{ \sum_{k=1}^{j} L_{i,k} \overline{C_j} \right\} \geq \max_i \left\{ \sum_{k=1}^{j} L_{i,k} \overline{C_k} \right\} \geq \max_i \left\{ \frac{(\sum_{k=1}^{j} L_{i,k})^2}{2} \right\}$$

which is equivalent to

$$\overline{C_j} \geq \frac{1}{2} \max_i \left\{ \sum_{k=1}^{j} L_{i,k} \right\} = \frac{1}{2} \Delta(\bigcup_{k \leq j} G_k)$$

□

2.7.1 Proof of the LP Rounding Version of the Main Theorems

Theorem 4. *There exists a deterministic, polynomial time 4-approximation algorithm for coflow scheduling without release times.*

Proof. Consider any coflow j and let $C_j(alg)$ denote the completion time of coflow j when scheduled as per Algorithm 3. Since all coflows have zero release times, at time $t_1 = 0$ all the coflows are arrived. Let $G'_1, \ldots, G'_n$ denote the coflows after all the edges have been moved backward. According to Lemma 1 each coflow G'_k could be finished at time $\Delta(G'_k)$, thus when the coflows are scheduled sequentially, we get

the following.

$$C_j(alg) = \sum_{k \le j} \Delta(G'_k)$$

Applying Lemma 2 and Lemma 7:

$$C_j(alg) = \sum_{k \le j} \Delta(G'_k) \le 2\Delta(\bigcup_{k \le j} G_k) \le 4\overline{C_j}$$

Hence, the total weighted completion time of our schedule can be bounded by the objective of the optimal LP solution.

$$\sum_{j=1}^{n} w_j C_j(alg) \le 4 \sum_{j=1}^{n} w_j \overline{C_j} \le 4OPT$$

□

Theorem 5. *There exists a deterministic, polynomial time 5-approximation algorithm for coflow scheduling with release times.*

Proof.

$$C_j(alg) \le \max_{k \le j} r_k + 2\Delta(\bigcup_{k \le j} G_k) = \max_{k \le j} r_k + 4\overline{C_j} \le 5\overline{C_j}$$

The first inequality follows from Lemma 4 and the second equality follows from Lemma 7. The last inequality holds since $\overline{C_j} \ge \overline{C_k}$ for all $1 \le k \le j$ and $\overline{C_k} \ge r_k$.

The cost of obtained coflow schedule is

$$\sum_{j=1}^{n} w_j C_j(alg) \le 5 \sum_{j=1}^{n} w_j \overline{C_j} \le 5OPT.$$

□

2.8 A Combinatorial 3-approximation Algorithm For Concurrent Open Shop with Release Times

Theorem 1. *Algorithm 1 gives a 3-approximation for concurrent open shop scheduling with release times.*

Proof. We use algorithm 1 to get a permutation $\{1, 2, \cdots, n\}$ for a set of jobs J. If we schedule the jobs according to this permutation sequentially, we'll get:

$$C_j \leq \max_{i' \leq j} r_{i'} + \sum_{k \leq j} L_{\mu(j),k}$$

Lemma 5 with $a = 1$ and $b = 1$, imply that:

$$\sum_j w_j C_j(alg) \leq (1 + \frac{1}{\kappa}) \sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + 2(\kappa + 1) \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S)$$

To minimize the approximation ratio, we substitute $\kappa = \frac{1}{2}$ and obtain

$$\sum_j w_j C_j(alg) \leq 3 \left(\sum_{j=1}^{n} \sum_{i \in M} \alpha_{i,j} r_j + \sum_{i \in M} \sum_{S \subseteq J} \beta_{i,S} f_i(S) \right) \leq 3 \cdot OPT$$

where the last inequality follows from weak duality as α and β constitute a feasible dual solution. □

2.9 Correction of Algorithm by Qiu et al.

We now give a brief overview of the approximation algorithm given by Qiu et al. [13].

2.9.1 Interval-Indexed LP Formulation

In the first step we write an interval-indexed linear programming relaxation for the coflow scheduling problem similar to that for the concurrent open shop problem by Wang and Cheng [35].

Let $\bar{C}_j$ denote the approximated completion time of coflow j obtained by an optimal feasible solution to this LP relaxation. We first order the coflows in non-decreasing order of these approximated completion times, i.e. we have the following.

$$\bar{C}_1 \leq \bar{C}_2 \ldots \leq \bar{C}_n \tag{2.19}$$

Let V_j denote the maximum load on any port by the *first* j coflows taken together in the above ordering, i.e.

$$V_j = \max \left[\max_i \left\{ \sum_{k=1}^{j} \sum_o d_{io}^k \right\}, \max_o \left\{ \sum_{k=1}^{j} \sum_i d_{io}^k \right\} \right].$$

Qiu et al. [13] prove that these V_j values provide a good approximation for the optimal completion times of the coflows. In particular, they show the following where C_j^* denotes the completion time of coflow j in an optimal schedule.

$$\sum_j w_j V_j \leq \frac{16}{3} \sum_j w_j C_j^* \tag{2.20}$$

2.9.2 Grouping Coflows

Divide time into geometrically increasing intervals as follows - $[1], [2], [3, 4], [5, 8], [9, 16], \ldots$. Let $I_l = (2^{l-2}, 2^{l-1}]$ denote the lth interval.

Now group the coflows based on the interval where their V values lie and let S_l denote the set of coflows assigned to interval I_l. In other words, all coflows $j \in S_l$, we have $2^{l-2} < V_j \leq 2^{l-1}$.

Algorithm 1

- For $l = 1, 2, \ldots$
  - Wait until the last coflow in S_l is released.
  - Group all coflows in S_l and schedule as per Algorithm 1 in [13]. This would take time at most $V_k \leq 2^{l-1}$ where k is the last job in the group.

Analysis

Qiu et al. claim the following (Proposition 1 in [13]).

Proposition 1. *For any coflow j, let $C_j(alg)$ denote the completion time of coflow j as per Algorithm 1. Then we have*

$$C_j(alg) \leq \max_{1 \leq g \leq j} \{r_g\} + 4V_j.$$

Since $C_j^* \geq \max_{1 \leq g \leq j}\{r_g\}$, Proposition 1 and Equation (2.20) together imply the following theorem (Theorem 1 in [13]).

Theorem 1. *There exists a deterministic polynomial time* 67/3 *approximation algorithm for coflow scheduling, i.e.*

$$\sum_j w_j C_j(alg) \leq \frac{67}{3} \sum_j w_j C_j^*.$$

2.9.3 Error

We now show that the Proposition 1 stated above is incorrect. Consequently, Theorem 1 no longer holds. Recall that Algorithm 1 groups jobs based on their V values alone and does not consider their release times.

Consider a simple case where $m = 1$ and we have just one input port and one output port. Say we have two jobs j_1 and j_2 such that j_1 needs to send 3 units of data and j_2 needs to send 1 unit of data. Also say $r_{j_1} = 0$ and $r_{j_2} = 100$. By definition, we have $V_{j_1} = 3$ and $V_{j_2} = 4$; note that both the jobs belong to the same interval $I_3 = (2, 4]$. Now since both jobs belong to the same interval, Algorithm 1 waits for both the jobs to be released and then schedules them together (after time 100). In this case, the claim in Proposition 1 clearly does not hold for job j_1.

Proposition 2 in [13] makes a similar claim for a grouping algorithm using randomized intervals. Again, the above instance serves as a counterexample to the claim. Consequently, Theorem 2 in [13] does not hold.

In the following section, we show that the deterministic grouping algorithm

can be modified to yield a $\frac{76}{3}$-approximation algorithm. Note that this is worse than the $\frac{67}{3}$ factor claimed earlier. It is not immediately clear whether the randomized algorithm from [13] can be corrected via a similar modification.

2.9.4 Corrected Grouping Algorithm

We first solve the interval-indexed LP formulation to obtain approximated completion times $\bar{C}_j$. Without loss of generality, we assume that the coflows are ordered as per Equation (2.19).

As shown by Leung, Li, and Pinedo (Theorem 13 in [33]), the analysis of Wang and Cheng [35] can be extended to the case of general release times to obtain the following.

$$\sum_j w_j \bar{C}_j \leq \frac{19}{3} \sum_j w_j C_j^* \tag{2.21}$$

This is analogous to Lemma 3 in [13] that shows that $\sum_j w_j V_j \leq \frac{16}{3} \sum_j w_j C_j^*$ where V_j is the maximum load on any port by the *first* j coflows taken together (as per the ordering).

Since $\bar{C}_j$ denotes the approximation completion time of coflow j as computed by the valid LP relaxation, we also have the following where r_j denotes the release time of coflow j.

$$\bar{C}_j \geq r_j \tag{2.22}$$

$$\bar{C}_j \geq V_j \tag{2.23}$$

2.9.4.1 Algorithm

Divide time into geometrically increasing intervals as follows - $[1], [2], [3,4], [5,8], [9,16], \ldots$. Let $I_l = (2^{l-2}, 2^{l-1}]$ denote the lth interval.

Now group the coflows based on the interval where their $\bar{C}$ values lie and let S_l denote the set of coflows assigned to interval I_l. So for all coflows $j \in S_l$, we have $2^{l-2} < \bar{C}_j \leq 2^{l-1}$.

Algorithm

- For $l = 1, 2, \ldots$
  - Wait until the last coflow in S_l is released AND all coflows in S_{l-1} have finished. (whichever is later).
  - Group all coflows in S_l and schedule as per Algorithm 1 in [13]. This would take time at most $V_k \leq 2^{l-1}$ where k is the last job in the group.

Analysis

Let $\tilde{C}_l$ denote the time by which all coflows in S_l have been scheduled by the above algorithm.

Claim 1. *$\tilde{C}_l \leq 2 \times 2^{l-1} = 2^l$ for every group S_l.*

Proof. We prove by induction. For group S_1, we start executing the schedule at $\max_{j \in S_1} r_j \leq \max_{j \in S_1} \bar{C}_j \leq 2^{1-1} = 1$ and the schedule takes time at most $V_k \leq 2^{1-1} = 1$ where k is the last coflow in the group. So the base case is true.

Now assume that the claim is true for some group S_l. As per the algorithm, the coflows in group S_{l+1} start executing at $\tilde{C}_l$ or $\max_{j \in S_{l+1}} r_j$ whichever is later. By induction, we are guaranteed that $\tilde{C}_l \leq 2^l$. Also $\max_{j \in S_{l+1}} r_j \leq \max_{j \in S_{l+1}} \bar{C}_j \leq 2^l$. Thus the coflows in group S_{l+1} start executing latest at time 2^l. We know that all these coflows require at most $V_k \leq \bar{C}_k \leq 2^l$ time units to complete. As a result, all the coflows in this group are scheduled by time $2^l + 2^l = 2^{l+1}$.

And thus the claim follows by induction. □

Claim 2. *For any coflow j, let $C_j(alg)$ denote the completion time of coflow j as per the algorithm. Then $C_j(alg) < 4\bar{C}_j$.*

Proof. Consider any coflow j, and let l be such that $j \in S_l$. Hence we have $\bar{C}_j > 2^{l-2}$. By the previous claim, we have

$$C_j(alg) \leq \tilde{C}_l \leq 2^l = 4 \times 2^{l-2} < 4\bar{C}_j$$

□

Corollary 2. *There is a deterministic $\frac{76}{3}$-approximation for coflow scheduling with arbitrary release times.*

Proof. Claim 2 and Equation (2.21) together imply a $\frac{76}{3}$-approximation algorithm for coflow scheduling with release times. □

2.10 Counterexample to Claim by Luo et al.

Luo et al. [7] claim a 2-approximation algorithm for the coflow scheduling problem by proving that it is equivalent to concurrent open shop scheduling. One of the key ingredients of their proof is the following claim that is implicit in Lemma 3 in Luo et al. [7].

Claim 3 (Restated from [7])**.** *Given two coflows G_k and G_l, we can find a feasible schedule for both the coflows such that* $C_k + C_l = \min\{\Delta(G_k) + \Delta(G_k \bigcup G_l), \Delta(G_l) + \Delta(G_k \bigcup G_l)\}$.

Counterexample

We show that Claim 3 is erroneous via a simple counterexample. Consider two coflows on a 3×3 datacenter as shown in Figure 2.4. Note that while coflows G_1 and G_2 have $\Delta(G_1) = 1$ and $\Delta(G_2) = 2$, the combined coflow $G_1 \bigcup G_2$ also has $\Delta(G_1 \bigcup G_2) = 2$. Consequently, the RHS in Claim 3 equals $\Delta(G_1) + \Delta(G_1 \bigcup G_2) = 3$.

On the other hand, as seen in Figure 2.4, if coflow G_1 is scheduled so that $C_1 = \Delta(G_1) = 1$, then the matching constraints force coflow G_2 to have completion time $C_2 = 3$. On the other hand, delaying one edge of coflow G_1, leads to a schedule with $C_1 = C_2 = 2$. In both cases, we have $C_1 + C_2 = 4$ (instead of 3) leading to a contradiction to the claim.

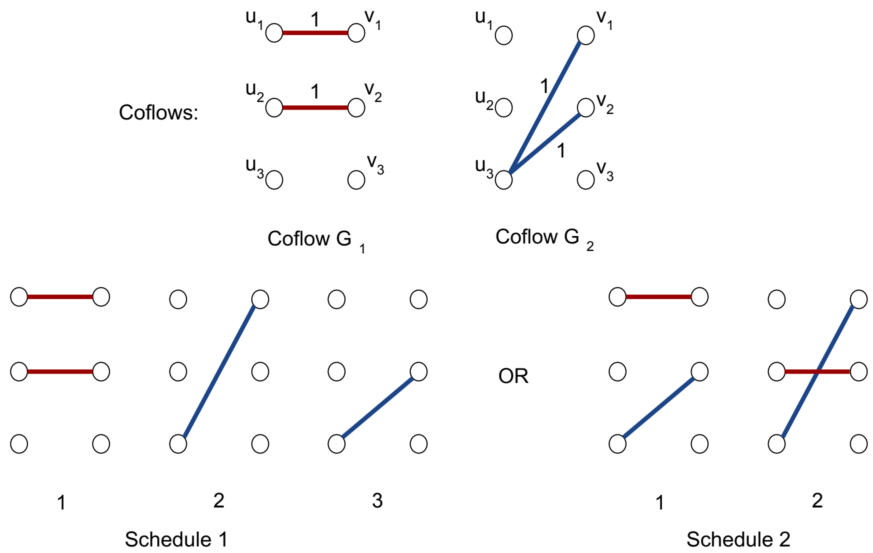

Figure 2.4: Simple counterexample to Claim 3

Chapter 3: Coflow Scheduling in Networks

3.1 Introduction

Modern computing applications have rather intensive computational needs. Many machine learning applications require up to tens of thousands of machines and often involve processing units across multiple data centers collaborating on the same application. This collaboration is usually handled by a large-scale distributed computing framework that ideally ensures a close-to-linear speedup compared to a single machine. A crucial part of the collaboration is that large chunks of data require both inter and intra-datacenter transmissions.

For intra-datacenter transmission, a common example would be the MapReduce framework. Map workers write all intermediate results independently to several servers to guard against failure and allow possible re-calculation. These results are shuffled and sent to Reduce workers. The volume of transmission between machines is so large that it has become a major bottleneck in the performance. In addition to this challenge, multiple applications may share the same cluster, and an uncoordinated schedule of their data transmission may cause an unacceptable delay in their completion times.

Chowdhury and Stoica [1] first introduced the abstraction of *coflow scheduling*,

which assumes that each application consists of a set of flows, and is finished once all the flows are completed. In their framework the network between machines is modeled as a switch: the input ports of different machines on one side, and output ports on the other side. A machine can send (receive) data to (from) any other machine, but to (from) only one machine at a time (sending and receiving may happen concurrently). The transmission speed between all machines is uniform. This describes a "perfect" datacenter where networking between machines is handled by a high-speed central switch (modeled by a complete bipartite graph) connected directly to all the machines [1]. However, real world datacenters are far more complicated; direct (virtual) links between machines may exist to avoid latency, duplicate links may exist to tolerate failure, network speeds may vary widely for different machines and links, and complicated network structures may exist for a variety of reasons. To make things worse, some tasks may involve multiple datacenters around the globe, and the switch model simply cannot accurately capture the graph based network that connects all the data centers.

For inter-datacenter transmission, distributed machine learning tasks can generate huge amounts of traffic. Due to legal or cost reasons, some datasets cannot be gathered into a single datacenter for processing. Instead, several geographically distributed datacenters work together to train a single model, and exchange local updates frequently to ensure accuracy and convergence. Though the size of a single transmission may be small considering the network bandwidth, the repeated exchange blows up the volume of transmission and makes network traffic its bottleneck.

In order to solve these problems, a slightly different model of coflow scheduling was proposed by Jahanjou et al. [2], which assumes that the underlying connection between machines is an arbitrary graph rather than a complete bipartite graph. Each node can be a machine, a datacenter or an exchange point (switch, router, etc.), and an edge between two nodes represents a physical link between the two Internet infrastructures. When some data needs to be transmitted from one node to another, it needs to be transmitted along edges. Unlike in the switch model where only one packet can be sent at each time slot, data for multiple jobs is allowed to transfer on the same link at the same time, or in other words, shared traffic on links is allowed. The total volume of data transmission on a link however is bounded by the link bandwidth[1]. Jahanjou et al. [2] considered the model in which data has to travel along a single specified path. In addition to this model, we also consider the *free path* model which allows data to be split or merged at nodes to utilize the whole graph when transmitting the same piece of data as long as the capacity of each link is respected. This seems much more complicated in practice than a single path transmission, but modern distributed computation frameworks [17] allow this kind of fine-grained control on network routing and transfer rate, which makes the model realistic. See Figure 3.1 for a brief illustration of the two models. The formal definitions come in Section 3.2.

[1]One major challenge in the switch model is the node-wise I/O speed constraint. In order to capture this in the graph model, we can replace every datacenter with a gadget of two nodes. The first node has exactly the same neighbors and edges that the original node for the datacenter has, plus links from and to the second node. The second node is only connected to the first node, and is the true source and destination for all demands involving this datacenter. By setting capacity on the links between these two nodes, we can enforce I/O limit for the whole datacenter like in the switch model.

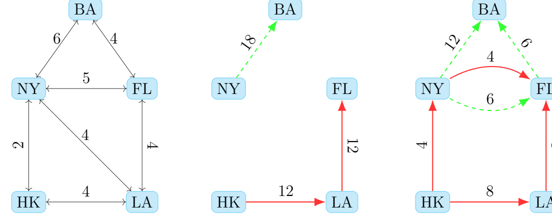

Figure 3.1: Example of coflow. The first graph shows the network topologies and the bandwidth of each link. We have one coflow consisting of two flows: one from NY to BA of demand 18 (denoted with dashed, green lines), the other from HK to FL of demand 12 (denoted with solid, red lines). The second graph shows the single path model, where each flow needs to be transmitted along a given path. It also implies a schedule in this model: transmit according to the path for 3 time units, and both flows are done. The third graph shows the free path model, where each flow can be split along multiple paths as long as the capacity of edges are respected. Here both flows can share the link from NY to FL and the entire coflow finishes in 2 units of time.

3.1.1 Related Works

We briefly describe the related works on coflow scheduling in networks here. For related works on the original coflow scheduling problem, please see Section 2.1. Zhao et al. [11] consider coflow scheduling over arbitrary graphs and attempt to jointly optimize routing and scheduling. They give a heuristic based on shortest job first, and use the idle slots to schedule flows from the longest job. Jahanjou et al. [2] studied two variants of coflow scheduling over general graphs, namely, when the path for a flow is given or if the path is unspecified. In both cases, the transmission rate may change over time, but each flow can only take a single path, whether given to or chosen by the fractional routing algorithm. In the first case, Jahanjou et al. [2] develop the first constant approximation algorithm (approximation ratio 17.6) and in the second case they develop an $O(\frac{\log n}{\log\log n})$ approximation algorithm (n is the number of nodes in the graph), matching the lower bound given by Chuzhoy et al.

[41].

3.1.2 Our Contributions

The main result of this chapter is a unified, tight 2-approximation algorithm for the coflow scheduling problem in both the single path model and the free path model when all release times and demands are polynomially sized, and a $(2+\epsilon)$-approximation when the release times and demands can be super-polynomial. This improves upon the 17.6 approximation given by Jahanjou et al. [2] for the single path model, and is the first approximation algorithm for the free path model (introduced by You and Chowdhury [17]).

We also evaluated our algorithm using two WAN topologies (Microsoft's SWAN [42] and Google's G-Scale [43]) on four different workloads (BigBench [44], TPC-DS [45], TPC-H [46], and Facebook (FB) [47, 48]) and compared with state-of-the-art for both models [2, 17]. For the single path model, we significantly improved over Jahanjou et al. [2]. For the free path model, we are close to what Terra [17] gets, but have the extra capability of dealing with weights. Across all variants and models, we have shown that taking the LP solution directly is an effective heuristic in practice.

3.1.3 Chapter Organization

In Section 3.2 we give a formal definition of the two models for coflow scheduling. In Section 3.3 we give a general linear program that deals with both models. We give the additional flow constraints for the two models in Section 3.3.1. In Sec-

tion 3.4.1 we describe the main algorithm and present the analysis in Section 3.4.2. We prove both models to be NP-hard in Section 3.5. In Section 3.6, we show experimental results by comparing our algorithms to some baseline algorithms.

3.2 Model and Problem Definition

We now formally define the models of coflow scheduling that we consider in this chapter. Let $G = (V, E)$ be a directed graph that represents the data center network and $c : E \rightarrow \mathbb{R}^+$ be a function that denotes the capacity (bandwidth) available on each edge of the network. Let $\mathcal{J} = \{F_1, F_2, \ldots, F_n\}$ denote the set of n coflows. A coflow F_j has weight w_j that denotes its priority and consists of n_j individual flows, i.e., $F_j = \{f_j^1, \ldots, f_j^{n_j}\}$ where $f_j^i = (s_j^i, t_j^i, \sigma_j^i)$ denotes a flow from source node $s_j^i \in V$ to sink $t_j^i \in V$ with demand $\sigma_j^i \in \mathbb{R}^+$. We assume that time is discrete and data transfer is instantaneous, i.e., it takes negligible time for data to cover multiple hops of edges as network delay is low compared to the time to transmit large chunks of data. A coflow F_j is said to be completed at the earliest time t such that for each flow $f_j^i \in F_j$, σ_j^i units of data have been transferred from source s_j^i to sink t_j^i. Our goal is to find a schedule that routes all the requisite flows (i.e. at any time, what fraction of a certain flow is transmitted and along which path/paths) subject to the edge bandwidth constraints so that the total weighted completion time of the coflows $\sum_j w_j C_j$ is minimized. Figure 3.2 gives an example of an instance of the coflow scheduling problem over a simple network.

We consider *two* different transmission models, based on whether a flow f_j^i has

restrictions as to how the data is transmitted. In the *single path model*, each flow f_j^i specifies a path p_j^i from source $s_j^i \in V$ to sink $t_j^i \in V$ so that the flow can only be routed along that path. This is exactly the "circuit-based coflows with paths given" model studied by Jahanjou *et al.* [2].

In the *free path model*, we can freely select the routing we desire for any flow f_j^i. In any time slot, data transmission occurs as a feasible multi-commodity flow so that both flow-conservation and edge bandwidth constraints are satisfied. Thus, we can split any flow f_j^i along multiple paths from its source to destination. This model was proposed in Terra [17]. Since the shortest paths of different flows can share edges and cause congestion, the free path model offers the flexibility of rerouting flows along less congested paths. In addition, modern internet infrastructures support using multiple paths together to get a higher overall speed (known as link aggregation), which is captured in the free path model as network flow.

In fact, both models are handled *uniformly* by the same framework, and the only difference is the set of flow constraints that describe what are considered feasible transmissions. It is also possible to handle other kinds of transmissions, like an intermediate case between single path and free path: several paths are given, and we can use them together and decide at what rate we are transmitting along each path. Figures 3.3 and 3.4 show the optimal solutions for the example coflow problem in the single path and free path models respectively.

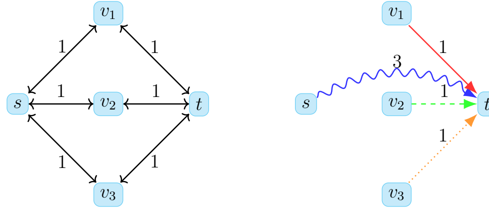

Figure 3.2: On the left is the graph structure: bi-directed edge of independent capacity of 1, on the right is the demanded coflow. There are four coflows each containing one single flow: red (solid) from v_1 to t, green (dashed) from v_2 to t, orange (dotted) from v_3 to t, and blue (curly) from s to t. The first three have demand 1, while the blue coflow has a demand of 3. All of them have the same weight of 1.

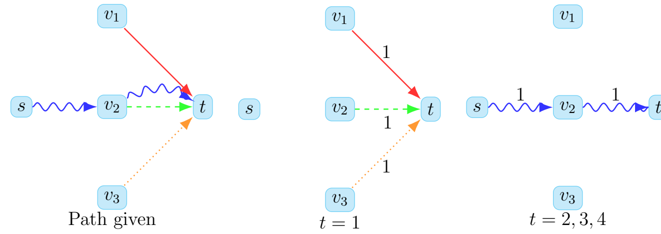

Figure 3.3: For the *single path* model, we have the path assignment in the left figure. Notice the path for green (dashed) flow shares an edge with that for the blue (curly) flow. Here is one optimal solution for the *single path model*. The total weighted completion time is $1 + 1 + 1 + 4 = 7$.

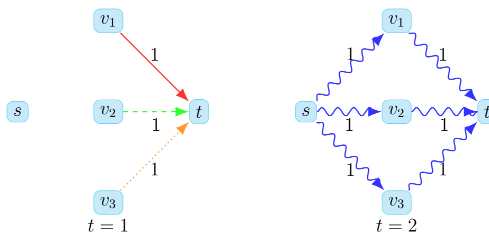

Figure 3.4: This is the optimal solution in the *free path model*. At time 1, send the red (solid), green (dashed), and orange (dotted) coflows. At time 2, send the blue (curly) coflow on all paths. The total weighted completion time is $1+1+1+2 = 5$.

3.3 Linear Programming Relaxation

We use a time-indexed linear program to model this problem. Let T denote an upper bound on the total time required to schedule all the coflows. Note that T might be super-polynomial if the release times or coflow sizes are large. However, there is a standard technique that achieves polynomial size at the cost of a $(1+\epsilon)$ factor on approximation ratio. We will assume T to be polynomial in the main part, and present the fix for super-polynomial T in Section 3.7.

Let time be slotted and time slot t cover the interval of time $[t-1, t]$. For a given flow f_j^i and a time slot t, we introduce the variable $x_j^i(t)$ to indicate the fraction of flow f_j^i that is scheduled at time t. For each coflow F_j, we introduce variables $X_j(t)$ to indicate if all the flows $f_j^i \in F_j$ have been completely scheduled by time t. Finally, we introduce a variable C_j that models the completion time of coflow F_j.

To make the linear program compatible with both single path model and free path model, we exclude the flow constraints and edge bandwidth constraints for now and delay them to Section 3.3.1.

$$\text{Minimize} \quad \sum_j w_j C_j, \qquad \text{subject to}$$

$$\sum_t x_j^i(t) = 1 \qquad \forall j \in [n], \forall i \in [n_j] \tag{3.1}$$

$$X_j(t) \leq \sum_{\ell=1}^{t} x_j^i(\ell) \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in T \tag{3.2}$$

$$C_j \geq 1 + \sum_t (1 - X_j(t)) \qquad \forall j \in [n] \tag{3.3}$$

$$r_j^i \geq t \Rightarrow x_j^i(t) = 0 \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in T \tag{3.4}$$

$$x_j^i(t) \geq 0 \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in T \tag{3.5}$$

Constraint (3.1) certifies that each flow is fully scheduled. Constraint (3.2) ensures that coflow F_j is considered completed at time t only if all flows $f_j^i \in F_j$ have been fully scheduled by time t. In Proposition 2, we show that Constraint (3.3) enforces a valid lower bound on the completion time of coflow F_j. Finally, Constraint (3.4) ensures that no flow is scheduled before it has been released. Note this is not a typical LP relaxation, since any fractional solution is valid. The main relaxation is around the completion time, since representing the exact completion time of job is beyond the capability of a linear program.

Proposition 2. *The completion time of a coflow F_j can be lower bounded by $C_j \geq 1 + \sum_t (1 - X_j(t))$ where $X_j(t) \in [0, 1]$ denotes the fraction of coflow F_j that has been completed by (the end of) time slot t.*

Proof. Conventionally, in time-indexed linear programming relaxations, the completion time of a job j is lower bounded by the fractional completion time in the schedule, or $C_j = C_j \cdot \sum_{t=1}^{T} x_j(t) \geq \sum_{t=1}^{T} t \cdot x_j(t)$. In our setting, this corresponds to the constraint $C_j \geq \sum_t t \cdot x_j(t)$ where $x_j(t) = X_j(t) - X_j(t-1)$ denotes the fraction of coflow F_j that is scheduled during time slot t. The desired constraint in Eq (3.3) is exactly the same constraint rearranged in a format that is more convenient for

analysis.

$$
\begin{aligned}
C_j &\geq \sum_{t=1}^{T} t \cdot x_j(t) = \sum_{t=1}^{T} x_j(t) \sum_{\tau=1}^{t} 1 \\
&= \sum_{\tau=1}^{T} \sum_{t \geq \tau}^{T} x_j(t) = \sum_{\tau=1}^{T} \left(\sum_{t=1}^{T} x_j(t) - \sum_{t=1}^{\tau-1} x_j(t) \right) \\
&= \sum_{\tau=1}^{T} (1 - X_j(\tau - 1)) = \sum_{\tau=0}^{T-1} (1 - X_j(\tau)) = 1 + \sum_{\tau=1}^{T-1} (1 - X_j(\tau))
\end{aligned}
$$

□

3.3.1 Model-specific Constraints

3.3.1.1 Single Path Model

In the single path model, a flow f_j^i can only be routed along a specified path p_j^i. Thus, we do not need to make any routing decisions in the linear program and only need to ensure that edge bandwidths are respected.

$$
\sum_{p_j^i \ni e} x_j^i(t) \cdot \sigma_j^i \leq c(e), \qquad \forall e \in E, \forall t \in T \tag{3.6}
$$

Constraint (3.6) enforces that the total flow scheduled through edge e at any time slot t does not exceed the edge bandwidth. Constraints (3.1)-(3.6) thus form the complete linear programming relaxation for coflow scheduling in the single path model.

3.3.1.2 Free Path Model

In the free path model, the path for flow f_j^i is not specified. In fact, data can split and merge at vertices to utilize all possible capacity. We use variable $x_j^i(t,e)$ to denote the fraction of flow f_j^i transmitted through edge e in time slot t. Recall that we use $x_j^i(t)$ to denote the total fraction of flow f_j^i that is transmitted in time slot t. $\delta_{in}(v)$ (δ_{out}) represents the set of edges that comes in (out of) vertex v. Here are the flow conservation constraints we need.

$$\sum_{e\in\delta_{out}(s_j^i)} x_j^i(t,e) = x_j^i(t), \qquad \forall j\in[n], \forall i\in[n_j], \forall t\in T \tag{3.7}$$

$$\sum_{e\in\delta_{in}(t_j^i)} x_j^i(t,e) = x_j^i(t), \qquad \forall j\in[n], \forall i\in[n_j], \forall t\in T \tag{3.8}$$

$$\sum_{e\in\delta_{in}(v)} x_j^i(t,e) = \sum_{e\in\delta_{out}(v)} x_j^i(t,e), \qquad \forall j\in[n], \forall i\in[n_j], \forall t\in T, \quad \forall v\in V\backslash\{s_j^i, t_j^i\} \tag{3.9}$$

$$\sum_{j\in[n], i\in[n_j]} x_j^i(t,e)\cdot\sigma_j^i \le c(e), \qquad \forall t\in T, \forall e\in E \tag{3.10}$$

Constraints (3.7) and (3.8) enforce that the total fraction of flow f_j^i satisfied at time t over all the paths is exactly $x_j^i(t)$. Constraints (3.9) ensure flow conservation at all nodes other than source and sink. Constraints (3.10) guarantee that all edge bandwidths are satisfied at all time steps. Constraints (3.1)-(3.5) and (3.7)-(3.10) thus form the complete linear programming relaxation for coflow scheduling in the free path model.

Let C_j^* denote the completion time of coflow F_j in an optimal solution of the LP relaxation, and let $C_j(opt)$ denote the completion time of coflow F_j in the corresponding optimal integral solution. Thus, for both the models, we have

$$\sum_j w_j C_j^* \leq \sum_j w_j C_j(opt). \tag{3.11}$$

3.4 Approximation Algorithms

Let $x_j^i(t)$ denote the fraction of flow f_j^i that is scheduled at time step t in an optimal solution to the above LP. The LP constraints guarantee that this yields a feasible schedule to the coflow scheduling problem (in both the single path as well as the free path models). However, since the completion time of a coflow F_j is defined as the earliest time t such that all flows $f_j^i \in F_j$ have been completely scheduled, the true completion time of coflow F_j obtained in this scheduled is given by

$$C_j(LP\ Sched) = \max_i \{ \max_{t: x_j^i(t) > 0} [t] \}. \tag{3.12}$$

Unfortunately, this completion time $C_j(LP\ Sched)$ can be much greater than the completion time variable in the optimal LP solution C_j^*, and thus the obtained schedule is not a constant-approximate coflow schedule. For instance, consider a coflow F_j with only one flow ($n_j = 1$) and let the optimal LP solution set its schedule as follows $x_j^1(1) = 0.9, x_j^1(10) = 0.1$, and $x_j^1(t) = 0, \forall t \notin \{1, 10\}$. Now, the completion time variable in the optimal LP solution is $C_j^* = \sum_t t x_j^1(t) = 1.9$.

However, true completion time of the coflow F_j in such a schedule is $C_j(LP\ Sched) = 10 \gg C_j^*$.

To overcome the obstacle above, we propose the following algorithm called Stretch (see Section 3.4.1) that modifies the schedule obtained by the linear program so that the completion time of each coflow in the modified schedule can be compared with the completion time variable of the corresponding coflow in an optimal LP solution. The schedule "stretching" idea (also called 'slow-motion') used in our algorithm has been used before successfully in other scheduling contexts [49, 50, 51].

3.4.1 Stretch Algorithm

1. Solve the linear program in Section 3.3 and obtain a fractional optimal solution.

2. Let $\lambda \in (0, 1)$ be drawn randomly according to the p.d.f $f(v) = 2v$. We can verify that this is indeed a valid probability distribution.

3. Stretch the LP schedule by $\frac{1}{\lambda}$. This means that we schedule everything exactly as per the LP solution - but whatever LP schedules in the interval $[a, b]$, we will schedule in the interval $[\frac{a}{\lambda}, \frac{b}{\lambda}]$.

4. Once σ_j^i units of flow f_j^i have been scheduled, leave the remaining slots for f_j^i empty.

Figure 3.5 illustrates the key ideas of the algorithm. To help understand this algorithm, start with the simple case where we have a fixed $\lambda = 0.5$, in other words

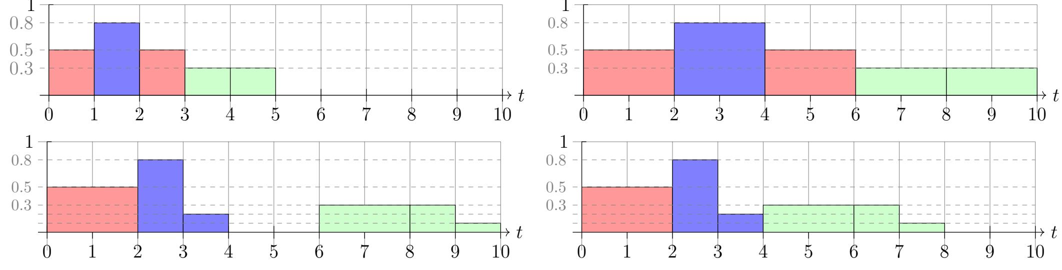

Figure 3.5: Here we show an example solution obtained from the LP, different color indicate different flows. In the second picture, we stretch with $\lambda = 0.5$. In the third picture, we leave the slots empty if the corresponding flow is finished. In the fourth picture, we utilize the idle slots and move some flows to earlier times. Though this does not improve the theoretically bound, it is beneficial in practice and is used in our experimental evaluation.

stretch the time axis by a factor of $1/\lambda = 2$. Intuitively, we move everything at time slot t and to both time slots $2t-1$ and $2t$. What used to be transmitted at time t will be transmitted no later than time $2t$. Consider any flow f_j^i and let τ denote the earliest time by which the LP has scheduled at least $1/2$ fraction of the flow. Then, it is easy to verify that the flow f_j^i is completely scheduled by time 2τ.

Now we consider a general λ and prove that this algorithm does output a feasible schedule. Due to fractional λ, it might be the case that some flow f_j^i of LP variable $x_j^i(t)$ in integral interval $[t-1, t]$ becomes $[\frac{t-1}{\lambda}, \frac{t}{\lambda}]$, a fractional interval. In this case, for a time slot τ, or a interval $[\tau - 1, \tau]$ after stretching, we just add $x_j^i(t) \cdot |[\tau - 1, \tau] \cap [\frac{t-1}{\lambda}, \frac{t}{\lambda}]|$.

The only flows that might be scheduled in time slot τ are those scheduled in time slot $1 + \lfloor \lambda(\tau - 1) \rfloor$ and $1 + \lfloor \lambda\tau \rfloor$ before stretching, or flows $f_j^i(1 + \lfloor \lambda(\tau - 1) \rfloor)$ and flows $f_j^i(1 + \lfloor \lambda\tau \rfloor)$. (The two time slots might be the same. If so, feasibility is automatically met. Otherwise, we have $1 + \lfloor \lambda(\tau - 1) \rfloor + 1 = 1 + \lfloor \lambda\tau \rfloor$.) For all flows at time $1 + \lfloor \lambda(\tau - 1) \rfloor$ before stretching, the factor we multiplied with

is $w_1 = \left|[\tau-1,\tau] \cap [\frac{\lfloor\lambda(\tau-1)\rfloor}{\lambda}, \frac{1+\lfloor\lambda(\tau-1)\rfloor}{\lambda}]\right|$. For all flows at time $1+\lfloor\lambda\tau\rfloor$ before stretching, the factor we use to multiply with is $w_2 = \left|[\tau-1,\tau] \cap [\frac{\lfloor\lambda\tau\rfloor}{\lambda}, \frac{1+\lfloor\lambda\tau\rfloor}{\lambda}]\right|$. Note $w_1 + w_2 = 1$. In fact, the schedule at time τ can be viewed as a weighted average of the schedule at time $[\lfloor\lambda(\tau-1)\rfloor, 1+\lfloor\lambda(\tau-1)\rfloor]$ and $[\lfloor\lambda\tau\rfloor, 1+\lfloor\lambda\tau\rfloor]$ (if $\lambda(\tau-1)$ is a integer, then the schedule will be exactly what it used to be at time $\lambda\tau$), the first with weight w_1 and the second with weight w_2. The nature of network flow ensures that the weighted sum of two feasible flows is a feasible flow.

Another fact that needs proof is that every flow is finished. This is guaranteed since schedules are stretched, and we only leave the remaining slots empty for f_j^i if σ_j^i units of flow have been scheduled, or in other words, all the demand for this flow has been scheduled.

3.4.2 Analysis

Recall that C_j^* denotes the completion time of coflow F_j in the optimal LP solution. While we consider that time is slotted in the LP formulation and time slot t covers the interval of time $[t-1, t]$, at this stage it is more convenient to work with continuous time rather than discrete time. For any continuous time $\tau \in [0, T]$, define $X_j(\tau)$ to be the fraction of coflow F_j that has been scheduled in the LP solution by time τ. We define $X_j(\tau)$ by assuming that the flow is scheduled at an uniform rate in every time slot. Formally, we have

$$X_j(\tau) = X_j(\lfloor\tau\rfloor) + (\tau - \lfloor\tau\rfloor)\left(X_j(\lfloor\tau\rfloor + 1) - X_j(\lfloor\tau\rfloor)\right). \tag{3.13}$$

The LP constraints (3.3) guarantee that for any coflow F_j, we have $C_j^* \geq 1 + \sum_t (1 - X_j(t))$. We can now lower-bound the LP completion time by replacing the above summation by an integral.

Lemma 1. $\int_{\tau=0}^{T} (1 - X_j(\tau)) d\tau \leq C_j^* - \frac{1}{2}$ *where* $X_j(\tau)$ *is defined as per Eq.* (3.13)*.*

Proof. By definition of $X_j(\tau)$, we have the following.

$$
\begin{aligned}
\int_{\tau=0}^{T} (1 - X_j(\tau)) d\tau &= T - \int_{\tau=0}^{T} X_j(\tau) d\tau \\
&= T - \sum_{t=0}^{T-1} \int_{\tau=t}^{t+1} X_j(\tau) d\tau \\
&= T - \sum_{t=0}^{T-1} \int_{\tau=t}^{t+1} \left[X_j(t) + (\tau - t) \left(X_j(t+1) - X_j(t) \right) \right] d\tau \\
&= T - \sum_{t=0}^{T-1} \left[X_j(t) + \left(X_j(t+1) - X_j(t) \right) \int_{\tau=t}^{t+1} (\tau - t) d\tau \right] \\
&= T - \sum_{t=0}^{T-1} \frac{1}{2} \left[X_j(t) + X_j(t+1) \right] \\
&= T - \left[\frac{1}{2} \left(X_j(0) + X_j(T) \right) + \sum_{t=1}^{T-1} X_j(t) \right]
\end{aligned}
$$

Since by definition, $X_j(0) = 0$ and $X_j(T) = 1$, we get

$$
= T - \left[\frac{1}{2} + \sum_{t=1}^{T-1} X_j(t) \right]
$$

Rearranging the terms, we get

$$
= 1 + \sum_{t=1}^{T-1} (1 - X_j(t)) - \frac{1}{2} \leq C_j^* - \frac{1}{2}
$$

where the last inequality follows from Constraint (3.3). □

For any $\lambda \in [0,1]$, define $C_j^*(\lambda)$ to be the earliest time τ such that λ fraction of the coflow F_j has been scheduled in the LP solution, i.e., in other words its the smallest τ such that $X_j(\tau) = \lambda$. Note that by time $C_j^*(\lambda)$, λ fraction of *every flow* $f_j^i \in F_j$ has been scheduled by the LP.

Proposition 3. $\int_{\lambda=0}^{1} C_j^*(\lambda)\mathbf{d}\lambda = \int_{\tau=0}^{T} (1 - X_j(\tau))\mathbf{d}\tau$

Proof.

$$\begin{aligned}
\int_{\lambda=0}^{1} C_j^*(\lambda)\mathbf{d}\lambda &= \int_{\lambda=0}^{1}\int_{\tau=0}^{T} \mathbb{1}_{[C_j^*(\lambda)>\tau]}\mathbf{d}\tau\mathbf{d}\lambda \\
&= \int_{\tau=0}^{T}\int_{\lambda=0}^{1} \mathbb{1}_{[C_j^*(\lambda)>\tau]}\mathbf{d}\lambda\mathbf{d}\tau \\
&= \int_{\tau=0}^{T}\int_{\lambda=X_j(\tau)}^{1} 1\mathbf{d}\lambda\mathbf{d}\tau = \int_{\tau=0}^{T}(1 - X_j(\tau))\mathbf{d}\tau
\end{aligned}$$

□

Finally, we are ready to bound the completion time of coflow F_j in the stretched schedule (denoted as $C_j(alg)$). For any fixed $\lambda \in (0,1)$, since we stretch the schedule by a factor of $\frac{1}{\lambda}$, we have $C_j(alg) \leq \left\lceil \frac{C_j^*(\lambda)}{\lambda} \right\rceil$. Notice the ceiling function in the bound [2]. Since λ is drawn randomly from a distribution, the following lemma bounds the expected completion time of coflow F_j in the stretched schedule.

Lemma 2. *The expected completion time of any coflow F_j in the stretched schedule is bounded by $2C_j^*$.*

[2] All flows $f_j^i \in F_j$ were completed by at least λ fraction by time $C_j^*(\lambda)$. So in the stretched schedule, all those flows must be completed by time $\frac{C_j^*(\lambda)}{\lambda}$. The ceiling is necessary since $\frac{C_j^*(\lambda)}{\lambda}$ may be fractional (i.e. occur in the middle of a time slot)

Proof.

$$\mathbb{E}[C_j(alg)] \leq \int_{\lambda=0}^{1} f(\lambda) \left\lceil \frac{C_j^*(\lambda)}{\lambda} \right\rceil d\lambda \leq \int_{\lambda=0}^{1} (2\lambda) \left(\frac{C_j^*(\lambda)}{\lambda} + 1 \right) d\lambda$$
$$= 2 \int_{\lambda=0}^{1} C_j^*(\lambda) d\lambda + 1$$

By Lemma 1 and Proposition 3,

$$= 2 \int_{\tau=0}^{T} (1 - X_j(\tau)) \mathbf{d}\tau + 1 \leq 2 \left(C_j^* - \frac{1}{2} \right) + 1 = 2C_j^*$$

□

Theorem 2 thus follows from the linearity of expectation.

Theorem 2. *There is a randomized 2-approximation algorithm for coflow scheduling in networks in both the single path and free path models when all release times and coflow sizes are polynomially sized.*

For the case where the total time we need to schedule all coflows is super-polynomial, we use the standard trick of geometric series time intervals, and claim the following theorem. Proof comes in Section 3.7.

Theorem 3. *For any $\epsilon > 0$, there is a randomized $(2 + \epsilon)$-approximation algorithm for coflow scheduling in networks in both the single path and the free path models (with possibly super polynomial release times and demands).*

3.5 Hardness of Approximation

We claim the following theorem:

Theorem 4. *For the coflow scheduling problem, in both the single path and the free path model, it is NP-hard to obtain a $(2-\epsilon)$ approximation, for any $\epsilon > 0$.*

Proof. We prove it by a reduction from concurrent open-shop problem (proved NP-hard to approximate within a factor better than $(2-\epsilon)$ [37, 38]). The definition of concurrent open shop problem is as follows: there are m machines and n jobs, each job j need to be processed on machine i for p_j^i time non-preemptively. We would like to minimize the total weighted completion time. Unlike the open shop problem, in the concurrent open shop problem a job can be processed on more than one machine at the same time.

Given a concurrent open-shop problem instance with M machines, we construct an instance of the coflow scheduling problem as follows. For every machine i, we have two nodes x_i and y_i, and an edge of unit bandwidth from x_i to y_i. Notice the graph has M different components, between each pair (x_i, y_i), there is only one path from x_i to y_i. Thus this construction works for both the single path model and the free path model. We will not distinguish the models in the following proof.

For a certain job j with demands σ_j^i in the concurrent open shop instance, we add a coflow j with demand of σ_j^i from x_i to y_i. Weights are directly taken from the concurrent open shop problem instance. Suppose we get a solution for this coflow scheduling instance, we can get a solution of no larger cost for the concurrent

open shop instance as follows. If we have a flow f_j^i for job j on edge (x_i, y_i) of size $x_j^i(t)$ at time t, then we schedule a fraction of $x_j^i(t)$ for job j on machine i at time t. Suppose a flow f_j^i is finished at time C_j^i in the coflow scheduling problem, the corresponding concurrent open shop problem for job j and machine i is also finished at time C_j^i. Similarly, the finishing time C_j of coflow j and concurrent open shop job j are the same. However, the solution we get is fractional, and might be preemptive (we might pause a job and resume it later).

Now we prove that we can modify this solution to get a non-preemptive integral solution without raising the total weighted completion time. For each machine i, consider all completion times C_j^i. Sort them in non-decreasing order $C_{l_1}^i, C_{l_2}^i, \ldots, C_{l_J}^i$, and we can safely reschedule these demand in the order of $l_1, l_2, \ldots, l_j$, and get new completion times $\mathbb{C}_{l_1}^i, \ldots, \mathbb{C}_{l_j}^i$ while not raising any completion time. We know all demand of job l_1 on machine i has been finished by $C_{l_1}^i$, so $\mathbb{C}_{l_1}^i = d_{l_1,i} \leq C_{l_1}^i$, similarly all demands of job l_1 and l_2 have been finished by $C_{l_2}^i$, and $\mathbb{C}_{l_2}^i = d_{l_1,i} + d_{l_2,i} \leq C_{l_2}^i$. We can continue and get $\mathbb{C}_{l_j}^i \leq C_{l_j}^i, \forall j \in \{J\}$. Thus the total weighted completion time for this integral solution would be upper bounded by the cost for the coflow scheduling instance.

$$\sum_{j \in \{J\}} w_j \cdot \mathbb{C}_j = \sum_{j \in \{J\}} w_j \cdot \max_{i \in \{M\}} \mathbb{C}_j^i \leq \sum_{j \in \{J\}} w_j \cdot \max_{i \in \{M\}} C_j^i = \sum_{j \in \{J\}} w_j \cdot C_j$$

For the other direction, for a certain solution of a concurrent open-shop problem, if task i of job j is scheduled from time t_1 to time t_2, we make the flow f_j^i take up all bandwidth of edge (x_i, y_i) from time t_1 to time t_2. Then flow f_j^i is finished

the same time when task i of job j is finished. Since every task i is finished the same time before and after reduction, completion times and the objective weighted completion time stays the same for the coflow scheduling problem.

In conclusion, for a solution SOL of concurrent open-shop problem with weighted completion time W, we can construct a solution SOL_{coflow} for coflow scheduling problem of the same weighted completion time W. For a solution SOL'_{coflow} of coflow scheduling problem with weighted completion time W', we can construct a solution SOL' for the original concurrent open-shop problem, with cost at most W'. Since concurrent open-shop problem is NP-hard to get a $(2-\epsilon)$ approximation, we know it is also NP-hard to approximate coflow scheduling problem to a factor of $(2-\epsilon)$, for both single path model and free path model. □

3.6 Experiments

We evaluated the Stretch Algorithm on 2 topologies and 4 benchmarks/industrial workloads. Experiments were run on a machine with dual Intel(R) Xeon(R) CPU E5-2430, and 64GB of RAM, and using Gurobi [52] as the LP solver. We first discuss the experimental set up and then in Section 3.6.2 discuss what evaluation we performed.

WAN topology: We consider the following graph topologies.

1. Swan [42]: Microsoft's inter-datacenter WAN with 5 datacenters and 7 inter-datacenter links. We calculate link bandwidth using the setup described by Hong et al. [42].

2. G-Scale [43]: Google's inter-datacenter WAN with 12 datacenters and 19 inter-

datacenter links.

Workloads: We use the following mix of jobs from public benchmarks - TPC-DS [45], TPC-H [46], and BigBench [44] - and from Facebook (FB) production traces [47, 48]. We follow [17] to set up the benchmarks: for a certain workload, jobs are randomly chosen and since they do not have a release time, we assign a release time similar to that in production traces. Each job lasts from a few minutes to dozens of minutes. Each benchmark experiment has 200 jobs. We randomly assign these jobs to nodes in the datacenter, and the demand will be between the corresponding nodes. Since weights are not available, we assign weights that are uniformly chosen from the interval between 1.0 and 100.0.

3.6.1 Implementation Details

In this subsection we discuss some details related to the implementation.

Time Index: There is a trade-off in selecting the size of a time slot. If the length of a time slot is shorter, we get more accurate answers, but need to solve a larger LP. On the contrary, if we make each time slot longer, the amount of computational resources need is greatly reduced, but the quality of the solution suffers. In all our experiments, we considered time slots of length 50 seconds as this led to tractable LP relaxations.

Rounding: Algorithm Stretch is meant for easy theoretical analysis, and is not a sophisticated rounding method; we are not trying to schedule later flows in the slots that are idle. This can cause huge overhead in experiments. See Figure 3.5 for an

illustration. In our implementation, we deal with this issue by moving the schedule of every time slot t to an earlier idle slot t' if for all flows scheduled at t, its release time is before t'.

To address the random sampling of λ, we sample 20 times from the distribution mentioned in Section 3.4.1 to get the expected weighted completion time for Algorithm Stretch, and denote it with "Average λ". We also measure the best solution obtained over these random choices (denoted by "Best λ").

3.6.2 Baselines

LP-based Heuristic: In addition to algorithms with theoretically worst case guarantee, we also propose a heuristic that works well in practice. Recall in Section 3.4.1, we mentioned that the LP solution itself is a valid schedule. We can use this solution as a heuristic, for both the single path and free path models. Note the weighted completion time for this LP solution is *NOT* the same as the LP objective function, as explained in Section 3.4.1. This implies that the solution from the heuristic can be arbitrarily bad in the worst case. In practice, however, this proves to be a very effective algorithm that can be quite close to the lower bound we get from LP.

Jahanjou et al. (*Single path model*): Since path information is not available in the datasets, we randomly generate one for each flow. For a source sink pair (s_j^i, t_j^i), we randomly select one of the shortest paths as the path for flow f_j^i. For this model, we compare our algorithm with the algorithm presented by Jahanjou et

al. [2]. Here is a brief description of their approach. First write an LP using geometric time intervals, then schedule each job according to the interval its α point (the time when α fraction of this job is finished) belongs to. A common reason for geometric time intervals is to avoid having a super-polynomial time horizon (a practical reason is to make the LP smaller), and a time series of $\{(1+\epsilon)^i\}$ is chosen where ϵ is close to 0. The closer ϵ is to 0, the better the approximation ratio can be. However, in Jahanjou et al.'s algorithm, the rounding step has a dependency on ϵ. To optimize the approximation ratio, ϵ is set to 0.5436. Our algorithm, on the contrary, is time slot based, and can be turned into a geometric series of time intervals by losing a factor of $(1 + \epsilon)$. In experiments, we include both the case of $\epsilon = 0.2$ and the case of $\epsilon = 0.5436$ for completeness.

Terra (*Free path model*): For the flow-based model, we are comparing to the offline algorithm in Terra [17]. This algorithm only works for the unweighted case. It calculates the time for each single coflow to finish individually, and then schedule with SRTF (shortest remaining time first). Instead of one large LP like all other algorithms compared here, this algorithm solves a large number of LPs, twice the number of coflow jobs. Terra can work with very fine grained time, to the order of milliseconds (and does not need time to be slotted). Since there is no previous work on weighted case, we compare the weighted case with the LP solution and our heuristic directly based on time indexed LP.

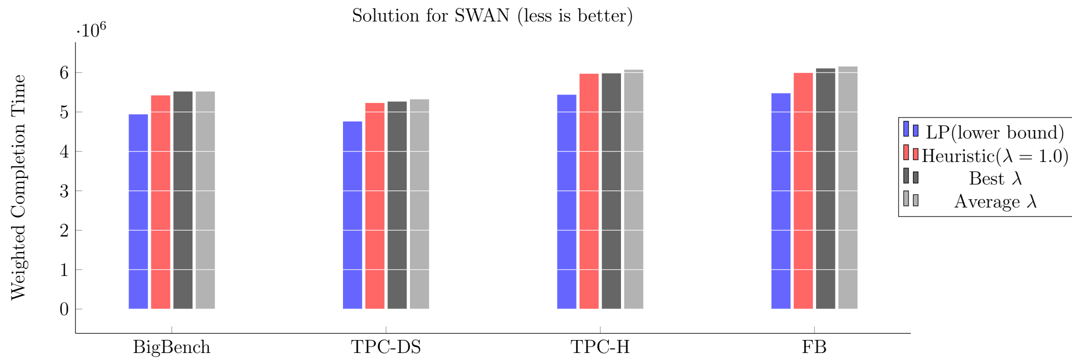

Figure 3.6: Free path model on SWAN, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1.

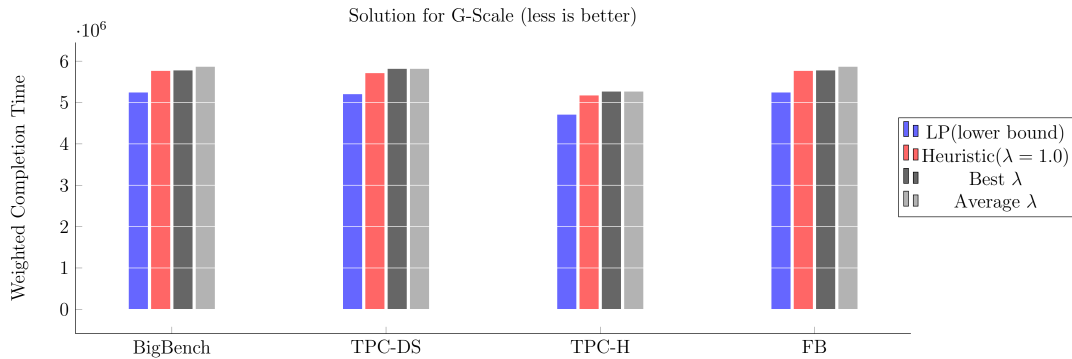

Figure 3.7: Free path model on G-Scale, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1.

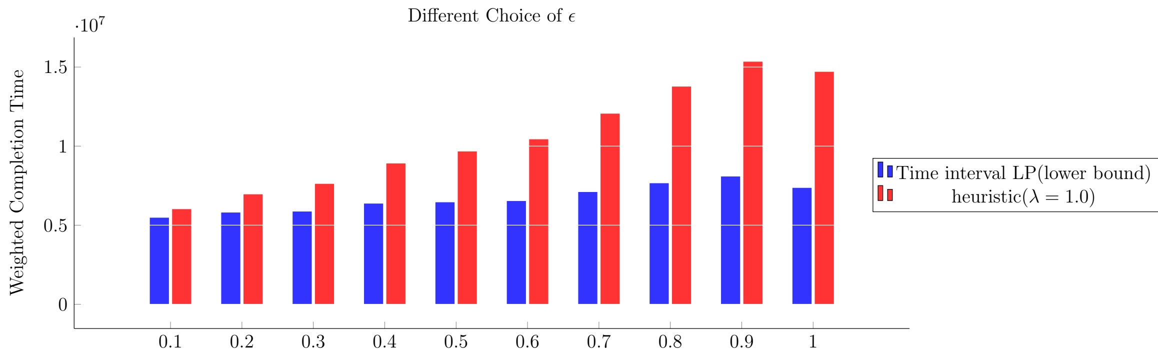

Figure 3.8: Free path model on SWAN for workload FB, the different choice of time interval ϵ may affect the performance bound of time interval LP value and the performance of heuristic ($\lambda = 1$).

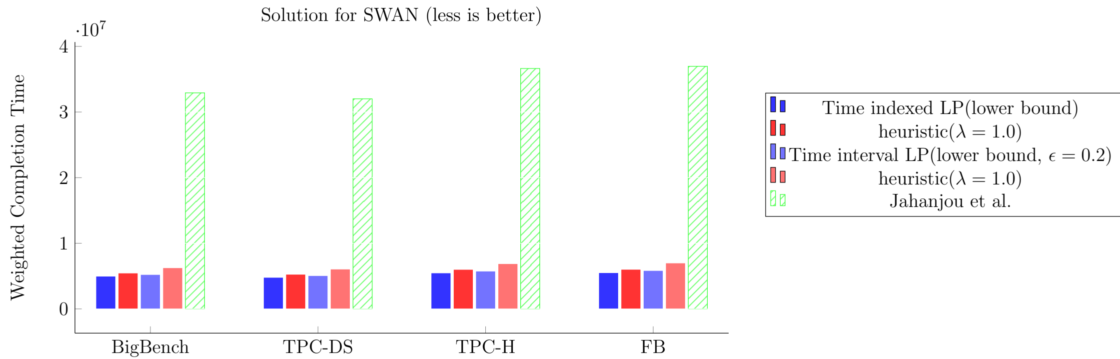

Figure 3.9: Single path model on SWAN, showing the performance bound of time indexed and time interval LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against algorithm by Jahanjou et al.[2].

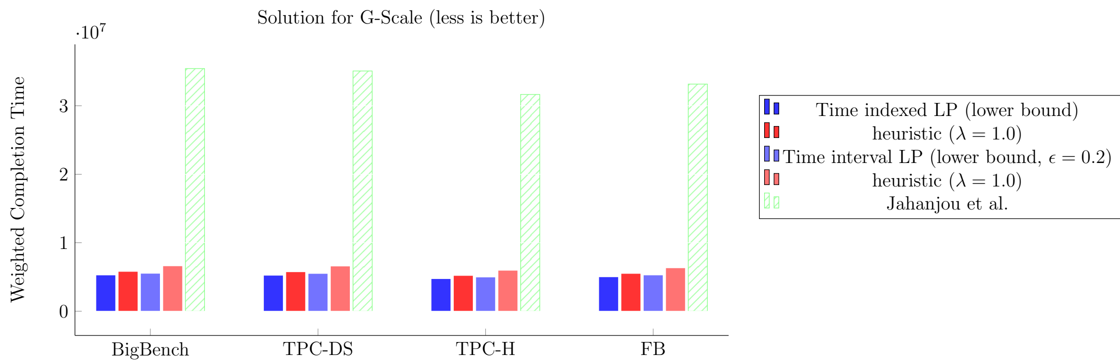

Figure 3.10: Single path model on G-Scale, showing the performance bound of time indexed and time interval LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against algorithm by Jahanjou et al.[2].

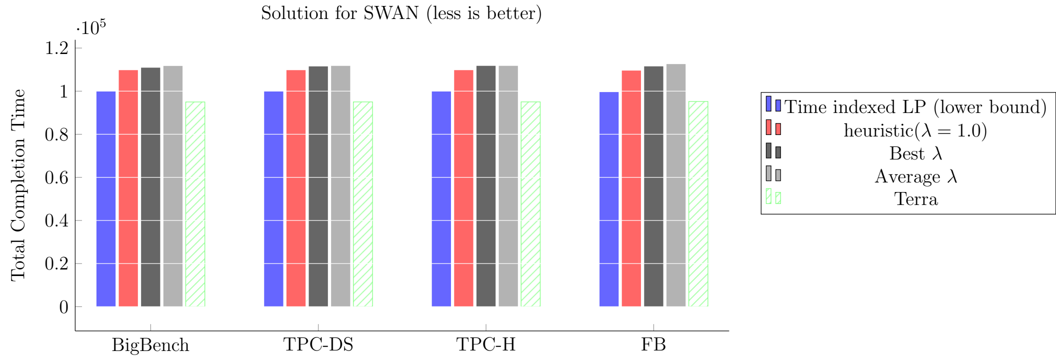

Figure 3.11: Free path model with no weight on graph SWAN, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against Terra[17]

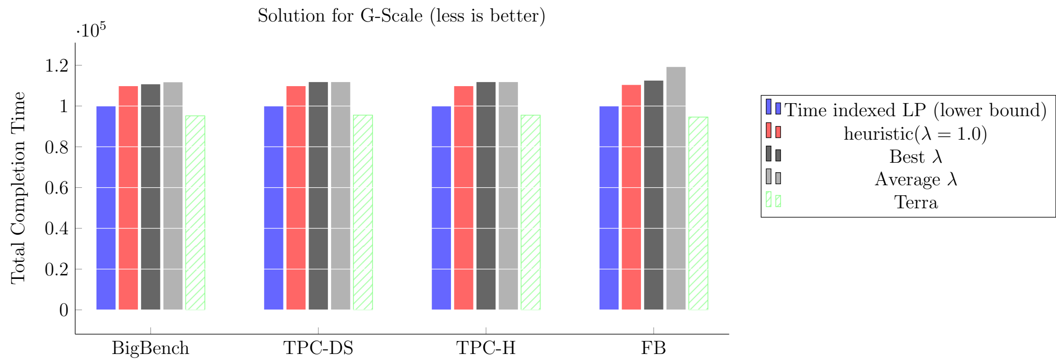

Figure 3.12: Free path model with no weight on graph G-Scale, showing the performance bound of time indexed LP value, the performance of heuristic ($\lambda = 1$), best λ among samples, and the expected value when λ is chosen from the distribution mentioned in Section 3.4.1. Here we compare against Terra[17].

3.6.3 Experimental Results

Impact of λ: See Figure 3.6 and Figure 3.7. When λ is 1.0, we take the LP solution directly (this is exactly the LP-based heuristic). Across all experiments, this seems the best choice of λ. The best sampled λ and the average case λ are pretty close, indicating the performance does not change much across different λ.

Impact of ϵ: To study the effect of the size of the time interval, we measure the LP objective and the schedule obtained by the LP-based heuristic as we vary ϵ in Figure 3.8. As ϵ increases, the size of the linear program will drop, making it faster to solve. On the other hand, the quality of solution drops, as we will not start a job until the whole current interval is after its release time, and will not consider a job finished until the interval its completion time belongs to ends. Thus a proper selection of ϵ may depend on the available computational resources for solving the LP.

Single Path Model: Figures 3.9 and 3.10 compare the performance of our algorithms with that of Jahanjou et al. [2] on all the benchmarks and topologies. Across all the experiments, we observe that our algorithms perform significantly better.

Free Path Model: See Figure 3.11 and Figure 3.12 for comparisons with the algorithm in Terra[17]. Since Terra only handles uniform coflow weights, we set all weights to be unit for these experiments. Surprisingly, we observe that Terra performs slightly better than even the LP objective itself. This disparity arises as the LP relies on time slots of 50 seconds while Terra deals with time slots of much

finer granularity. For the weighted case, we are not aware of previous work, and only compare to LP solution in Figures 3.6 and 3.7.

3.7 Sketch of generalization to super-polynomial time span

Geometric series time interval is defined as follows. For an $\epsilon > 0$, let $\tau_0 = 0, \tau_1 = 1, \cdots, \tau_k = (1+\epsilon)^{k-1}, \cdots$. We define the k-th interval as $l_k = [\tau_{k-1}, \tau_k]$. Since T is at most the sum of all processing time and all release time, we know the number of intervals $\mathbb{T} = 1 + \lceil \log_{1+\epsilon} T \rceil$ is polynomial.

We change the LP as follows. We abuse notation a bit and allow $\mathbb{T}$ to represent the set $\{1, 2, \cdots, \mathbb{T}\}$ when there is no confusion. We replace all accurance of T with $\mathbb{T}$ in Section 3.3, modify Equation (3.4) and Equation (3.3) to accommodate for release time, and get the following linear program.

$$\text{Minimize} \quad \sum_j w_j C_j, \qquad \text{subject to}$$

$$\sum_t x_j^i(t) = 1 \qquad \forall j \in [n], \forall i \in [n_j] \tag{3.14}$$

$$X_j(t) \leq \sum_{\ell=1}^{t} x_j^i(\ell) \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in \mathbb{T} \tag{3.15}$$

$$C_j \geq 1 + \sum_t (\tau_t - \tau_{t-1})(1 - X_j(t)), \qquad \forall j \in [n] \tag{3.16}$$

$$r_j^i \geq \tau_t \Rightarrow x_j^i(t) = 0 \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in \mathbb{T} \tag{3.17}$$

$$x_j^i(t) \geq 0 \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in \mathbb{T} \tag{3.18}$$

For the model specific part of linear program, we only need to change the capacity constraints: replace Equation (3.6) for single path model to get

$$\sum_{p_j^i \ni e} x_j^i(t) \cdot \sigma_j^i \leq (\tau_t - \tau_{t-1})c(e), \qquad \forall e \in E, \forall t \in \mathbb{T} \tag{3.19}$$

and Equation (3.10) for free path model to get

$$\sum_{e \in \delta_{out}(s_j^i)} x_j^i(t,e) = x_j^i(t), \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in \mathbb{T} \tag{3.20}$$

$$\sum_{e \in \delta_{in}(t_j^i)} x_j^i(t,e) = x_j^i(t), \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in \mathbb{T} \tag{3.21}$$

$$\sum_{e \in \delta_{in}(v)} x_j^i(t,e) = \sum_{e = \delta_{out}(v)} x_j^i(t,e), \qquad \forall j \in [n], \forall i \in [n_j], \forall t \in \mathbb{T}, \quad \forall v \in V \backslash \{s_i, t_i\} \tag{3.22}$$

$$\sum_{j \in [n], i \in [n_j]} x_j^i(t,e) \cdot \sigma_j^i \leq (\tau_t - \tau_{t-1})c(e), \qquad \forall t \in \mathbb{T}, \forall e \in E \tag{3.23}$$

Similar to Proposition 2, we prove Constraint (3.16) is a good lower bound.

Proposition 4. *The completion time of a coflow F_j can be lower bounded by $C_j \geq 1 + \sum_t (\tau_t - \tau_{t-1})(1 - X_j(t))$ where $X_j(t) \in [0,1]$ denotes the fraction of coflow F_j that has been completed by (the end of) time interval $[\tau_{t-1}, \tau_t]$.*

Proof. If a job completes in the interval $(\tau_{t-1}, \tau_t]$, then its finishing time is at least $\tau_{t-1} + 1$.

$$C_j \geq \sum_{t=1}^{\mathbb{T}} (1 + \tau_{t-1}) \cdot x_j(t) = \sum_{t=1}^{\mathbb{T}} x_j(t) + \sum_{t=1}^{\mathbb{T}} x_j(t) \sum_{\rho=1}^{t-1} (\tau_\rho - \tau_{\rho-1})$$

$$= 1 + \sum_{\rho=1}^{\mathbb{T}-1} (\tau_\rho - \tau_{\rho-1}) \sum_{t=\rho+1}^{\mathbb{T}} x_j(t)$$

$$= 1 + \sum_{\rho=1}^{\mathbb{T}-1} (\tau_\rho - \tau_{\rho-1}) \left(\sum_{t=1}^{\mathbb{T}} x_j(t) - \sum_{t=1}^{\rho} x_j(t) \right)$$

$$= 1 + \sum_{\rho=1}^{\mathbb{T}-1} (\tau_\rho - \tau_{\rho-1})(1 - X_j(\rho))$$

$\square$

After getting a solution, we would schedule coflows into intervals instead of into time slots. Inside each time interval, we just schedule each flow at uniform speed, and break into actual time slots. Similar to Section 3.4.1, we can prove that this solution is feasible.

3.7.1 Analysis

Recall that C_j^* denotes the completion time of the coflow F_j in the optimal LP solution. For any continuous time $t \in [0, T]$, define $\widehat{X}_j(t)$ to be the fraction of coflow F_j that has been scheduled in the LP solution by time t. Note $X_j(t)$ is for time interval $[\tau_{t-1}, \tau_t]$, but $\widehat{X}_j(t)$ is for original time slots. Flows are scheduled at an uniform rate in every time interval. Use $\rho(t)$ to denote the smallest ρ such that $t \in (\tau_{\rho-1}, \tau_\rho]$, we have

$$\widehat{X}_j(t) = X_j(\rho(t)) + \frac{t - \tau_{\rho(t)-1}}{\tau_{\rho(t)} - \tau_{\rho(t)-1}} \left(X_j(\rho(t) + 1) - X_j(\rho(t)) \right) \tag{3.24}$$

Similar to Lemma 1, we state and prove the following lemma.

Lemma 3. $\int_{t=0}^{T}(1-\widehat{X}_j(t))\mathbf{d}t \leq (1+\epsilon)C_j^* - \frac{1}{2}$ *where* $X_j(\rho)$ *is defined as per Eq. (3.24).*

Proof. From constraints (3.16), we have that

$$\int_{t=0}^{T}(1-\widehat{X}_j(t))\mathbf{d}t = \sum_{\rho=1}^{\mathbb{T}}\int_{t=\tau_{\rho-1}}^{\tau_\rho}(1-\widehat{X}_j(t))\mathbf{d}t$$

Since $1-\widehat{X}_j(t)$ is linear for $t \in (\tau_{\rho-1}, \tau_\rho]$,

$$\begin{aligned}
=&\sum_{\rho=1}^{\mathbb{T}}\frac{\tau_\rho-\tau_{\rho-1}}{2}(1-X_j(\rho)+1-X_j(\rho-1))\\
=&\sum_{\rho=1}^{\mathbb{T}}\frac{\tau_\rho-\tau_{\rho-1}}{2}(1-X_j(\rho))+\sum_{\rho=1}^{\mathbb{T}}\frac{\tau_\rho-\tau_{\rho-1}}{2}(1-X_j(\rho-1))\\
=&\sum_{\rho=1}^{\mathbb{T}}\frac{\tau_\rho-\tau_{\rho-1}}{2}(1-X_j(\rho))\\
&+(1+\epsilon)\sum_{\rho=2}^{\mathbb{T}}\frac{\tau_{\rho-1}-\tau_{\rho-2}}{2}(1-X_j(\rho-1))+\frac{\tau_1-\tau_0}{2}(1-X_j(0))\\
=&\sum_{\rho=1}^{\mathbb{T}-1}\frac{\tau_\rho-\tau_{\rho-1}}{2}(1-X_j(\rho))+\frac{\tau_{\mathbb{T}}-\tau_{\mathbb{T}-1}}{2}(1-X_j(\mathbb{T}))\\
&+(1+\epsilon)\sum_{\rho=1}^{\mathbb{T}-1}\frac{\tau_\rho-\tau_{\rho-1}}{2}(1-X_j(\rho))+\frac{\tau_1-\tau_0}{2}(1-X_j(0))\\
=&\frac{2+\epsilon}{2}\sum_{\rho=1}^{\mathbb{T}-1}(\tau_\rho-\tau_{\rho-1})(1-X_j(\rho))+\frac{1}{2}
\end{aligned}$$

Plugging in Proposition 4,

$$\leq\frac{2+\epsilon}{2}(C_j^*-1)+\frac{1}{2}\leq(1+\epsilon)C_j^*-\frac{1}{2}$$

□

For any $\lambda \in [0,1]$, define $C_j^*(\lambda)$ to be the earliest time ρ such that λ fraction of the coflow F_j has been scheduled in the LP solution, i.e., in other words its the smallest t such that $\widehat{X}_j(t) = \lambda$. Note that by time $C_j^*(\lambda)$, λ fraction of *every flow* $f_j^i \in F_j$ has been scheduled by the LP.

Proposition 5. $\int_{\lambda=0}^{1} C_j(\lambda)\mathbf{d}\lambda \leq \int_{t=0}^{T} (1 - \widehat{X}_j(t))\mathbf{d}t.$

Proof.

$$\int_{\lambda=0}^{1} C_j(\lambda)\mathbf{d}\lambda = \int_{\lambda=0}^{1}\int_{t=0}^{T} \mathbb{1}_{[C_j(\lambda)>t]}\mathbf{d}t\mathbf{d}\lambda = \int_{t=0}^{T}\int_{\lambda=0}^{1} \mathbb{1}_{[C_j(\lambda)>t]}\mathbf{d}\lambda\mathbf{d}t$$
$$= \int_{t=0}^{T}\int_{\lambda=X_j(t)}^{1} 1\mathbf{d}\lambda\mathbf{d}t = \int_{t=0}^{T} (1 - \widehat{X}_j(t))\mathbf{d}t$$

□

Finally, we are ready to bound the completion time $C_j(alg)$ of coflow F_j in the stretched schedule. For any fixed $\lambda \in (0,1)$, since we stretch the schedule by a factor of $\frac{1}{\lambda}$, it is easy to verify[3] that $C_j(alg) \leq \left\lceil \frac{C_j(\lambda)}{\lambda} \right\rceil$. Since λ is drawn randomly from a distribution, the following lemma bounds the expected completion time of coflow F_j in the stretched schedule.

Lemma 4. *The expected completion time of any coflow F_j in the stretched schedule is bounded by $2(1+\epsilon)C_j^*$.*

[3] All flows $f_j^i \in F_j$ were completed by at least λ fraction by time $C_j(\lambda)$. So in the stretched schedule, all those flows must be completed by time $\left\lceil \frac{C_j(\lambda)}{\lambda} \right\rceil$.

Proof.

$$\begin{aligned}\mathbb{E}[C_j(alg)] &\leq \int_{\lambda=0}^{1} f(\lambda)\left\lceil \frac{C_j(\lambda)}{\lambda}\right\rceil d\lambda \leq \int_{\lambda=0}^{1} 2\lambda\left(1+\frac{C_j(\lambda)}{\lambda}\right) d\lambda \\ &= \int_{\lambda=0}^{1} 2\lambda d\lambda + 2\int_{\lambda=0}^{1} C_j(\lambda) d\lambda \\ &= 1 + 2\int_{\lambda=0}^{1} C_j(\lambda) d\lambda\end{aligned}$$

By Lemma 3 and Proposition 5,

$$\leq 2(1+\epsilon)C_j^*.$$

□

Theorem 3 thus follows from the linearity of expectation.

3.8 Conclusion

In this chapter we developed an efficient approximation algorithm for the coflow scheduling problem in general graph topologies. This algorithm is shown to be practical and one that delivers extremely high quality solutions. The new insight was to write a time indexed LP formulation and to convert it using the idea of stretching the schedule.

The next major challenge is developing *online* methods for coflow scheduling to minimize weighted flow time. Prior work Khuller et al. [29] deals with the problem of minimizing weighted completion time by making use of offline approximation

algorithms. However, the problem of minimizing weighted flow time is considerably more challenging. The technical difference is that flow time is defined as $C_j - r_j$ where C_j is the completion time of a job, and r_j the release time. Optimizing flow time non-preemptively even on a single machine (a different model) is a notoriously difficult problem with some recent progress [53, 54].

Chapter 4: Scheduling with Spot Instances

4.1 Introduction

The rapidly increasing adoption of machine learning and cloud computing are arguably two of the most important technological trends of the last decade. While cloud computing has greatly simplified infrastructure management and software delivery for small and large businesses alike, machine learning has enabled software applications to improve in quality by learning from various sources of data. The intersection of these two trends is unavoidable today and it generates several questions of interest for the computer science community. The utility and novelty of this intersection stems from the unique characteristics of machine learning training jobs and the dynamics of the cloud computing market.

Cloud Computing Instance Characteristics: Cloud vendors can provide cheap computational resources owing to economies of scale. Businesses often provision cloud resources to sustain peak demands from critical services for their customers. However, the actual demands display large fluctuations across time and availability zones and may show specific diurnal and seasonal patterns [55, 56]. During times of low actual demand, cloud vendors make the unused resources (i.e. compute cores) available for use as cheaper entities that may be interrupted. These

are called *spot instances* by Amazon Web Services (AWS), *low-priority VMs* by Microsoft Azure, *preemptible instances* by Google Cloud and *transient* virtual machines/servers in literature [57, 58, 59, 60, 61]. In practice, spot instances[1] are often available at up to 70%-90% discounts compared to their on-demand equivalences [62, 63]. Data on *interruption frequency* for spot instances of different configurations in various availability zones is published by AWS [63] and it suggests that interruptions do occur in practice with non-negligible probability.

Machine Learning Characteristics: ML training is concerned with estimating parameters of an ML model family to produce a model instance that best fits a given dataset according to a given statistical performance metric. An ML training algorithm is often an iterative algorithm (each iteration is also known as an epoch) derived from the (stochastic) gradient descent family of algorithms [64, 65] and it produces better estimates of the model parameters with each iteration, usually with diminishing marginal returns. With the explosion in data availability, rising hardware capabilities and resurgence of deep learning [66] over the past decade, a practical trend in ML has been to use large model families to leverage larger datasets at the expense of longer training times. To train ML models in the ballpark of state-of-the-art statistical accuracy, it can take anywhere between a few days to a few months with today's hardware. If such a long-running ML training jobs is interrupted prematurely, model parameter estimates from the latest successfully completed iteration is still a valid model instance.

[1]Terminology: We refer to all such revocable computing instances as *spot instances*.

4.1.1 Training ML jobs on the Cloud

Cloud computing is convenient, attractive and often the go-to choice for developing and deploying machine learning powered software applications and services at scale. However, the cost of large scale ML training could be prohibitively high, especially if on-demand instances are used for ML training jobs. For a cloud customer that rents servers within her budget, she hopes to get as much computation power as possible. In order to achieve this, the low cost spot instance is a superior option if interruptions can be handled properly. In this paper, we try to answer the following question from a theoretical perspective:

> How can ML training jobs be scheduled and executed on interruptible but relatively inexpensive spot instances to increase their cost efficiency?

Here is the typical process of running a single ML training job on spot instances. You pick one out of all the available instances to dispatch the current job, possibly with a budget cap. The job will run either until an interruption happens or when the budget cap is met. If there is still budget remaining, we reschedule the job and resume from the last checkpoint. This process is repeated until all the budget is used up. The final utility, e.g. a constant minus the loss function, is a combination of all the progress you make across multiple reschedules. Typically, this utility is a sub-additive function of the number of epochs. If there are multiple jobs, you also need to balance the computation received for each job so the budget is spent on the most "profitable" jobs.

Several factors and trade-offs need to be accounted for before the aforemen-

tioned question can be comprehensively answered and a scheduling algorithm be designed. The first obstacle is that interruptions and restorations result in inevitable overheads when a job runs on spot instances. If the overhead is large enough and/or the interruption frequency is high enough, the overall time and hence the dollar cost for an ML training job on cheaper spot instances may overshoot that on an expensive interruption-free on-demand instance. We handle this trade-off between cheap price and uninterrupted computation by mapping it to the *correlated knapsack problem*. In this problem, items of stochastic states are packed into a capacitated bag. Each state of an item corresponds to a value and a weight, and the state is only revealed when it is added in the bag. We add items one by one into bag until it overflows, and gets the total value of all the items except the overflown one. Another layer of complexity is added by rescheduling. When a job is interrupted, we may decide to reschedule it on one (possibly the same) instance, which may in turn also be interrupted. Powerful techniques like dynamic programming will only work for unlimited supply of instances. The interruption of a configuration may indicate an imbalance in its demand and supply, and discourage us from dispatching (the same or other) jobs on it. There can also be a quota enforced by cloud providers, so we can use no more than a limited number of copy of a particular instance. This trade-off is captured by a partition matriod[2]. The last challenge is non-linearity. Typical ML training jobs have diminishing marginal returns, where twice the number of epochs will not give twice the utility. This can be handled easily if no rescheduling is allowed, but the combination of training epochs in separate reschedules is far from

[2]See definition in Section 4.2.3

trivial. The situation is further complicated when multiple jobs are involved. We use a variant of the monotone submodular function [3] to model the utility function with diminishing returns. In this work, we mainly focus on ML training that runs on a single machine. For the increasingly popular distributed ML training, it is not obvious how spot instances can be used. One attempt was Zhang et al. [67], in which a group of identical spot instances are rented and utilized in synchronization. Our model can handle this case by using meta instances corresponding to such bundles.

4.1.2 Our Contributions

We model the spot instance scheduling problem and map it to the correlated stochastic knapsack problem with a submodular target function. We present an algorithm that computes an adaptive policy for this problem which is guaranteed to achieve $(1-1/\sqrt{e})/2 \simeq 0.1967$ of the optimal solution on expectation. It improves on the $(1-1/\sqrt[4]{e})/2 \simeq 0.1106$ approximation algorithm from Fukunaga et al. [28]. Furthermore, we remove the assumption in Fukunaga et al. [28] where possible overflown of the budget is not allowed.

4.1.3 Our Techniques

If the target function is linearly additive, this problem becomes the correlated stochastic knapsack problem. For this problem, Gupta et al. [68] gave an 1/8 approximation algorithm for adaptive policies based on LP relaxation. The approxi-

[3] A function $f: 2^{\mathcal{N}} \to \mathbb{R}$ is *submodular* if for every $A \subseteq B \subseteq \mathcal{N}$ and $e \in \mathcal{N}: f(A \cup \{e\}) - f(A) \geq f(B \cup \{e\}) - f(B)$. An equivalent definition is that for every $A, B \subseteq \mathcal{N}: f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$.

mation ratio was improved to $1/(2+\epsilon)$ in Ma [69, 70], via a different LP formulation and a more sophisticate rounding scheme. A natural idea for the submodular target function case would be to generalize these algorithms, which is exactly what Fukunaga et al. [28] did. Their algorithm extends the 1/8 approximation algorithm in Gupta et al. [68], and achieves a $(1-1/\sqrt[4]{e})/2$ approximation. This is achieved by a combination of the stochastic continuous greedy algorithm [71], and the contention resolution scheme [72]. The $1/(2+\epsilon)$ algorithm by Ma [70] is based on a different and tighter LP, and the rounded solution exhibits complicate dependencies. Worse still, their rounding scheme would break a monotone property, which is a critical component of the contention resolution scheme. This rules out the possibility of a direct merger of the two algorithms. We manage to overcome this obstacle by a direct analysis of the correlated probability of events to fit in the contention resolution scheme. A factor of $(1-1/\sqrt{e})$ is lost for the continuous optimization part, and another factor of 2 is lost for rounding, leading to our $(1-1/\sqrt{e})/2 \simeq 0.1967$ approximation algorithm.

4.2 Problem Formulation

4.2.1 Notations

For a given integer $n \in \mathbb{Z}^+$, let $[n] = \{0, 1, 2, \ldots, n\}$. Let S be a set of items. Given two vectors $u, v \in [n]^S$, $u \leq v$ denotes the coordinate wise inequality, i.e. $\forall i \in S, u(i) \leq v(i)$. $u \vee v$ and $u \wedge v$ are also defined coordinate wise: $(u \vee v)(i) = \max\{u(i), v(i)\}$, $(u \wedge v)(i) = \min\{u(i), v(i)\}$. For a function $f : [M]^n \to \mathbb{R}^+$, it is

called *monotone* if $f(u) \leq f(v)$ for all $u \leq v$, and is called lattice-submodular if $f(u) + f(v) \geq f(u \wedge v) + f(u \vee v)$ holds for all $u, v \in [M]^n$.

4.2.2 Scheduling with Spot Instances

We model the problem as follows. $\mathcal{N}$ jobs need to be scheduled on $\mathcal{M}$ different instances, where the instances may have different CPU/RAM configurations or come from different available zones, i.e. have different interruption patterns. We assume each instance has a finite supply, the prices/interruptions of different instances and different copies of the same instance are independent. For a spot instance i, the length of time a job can run on it before interruption follows a known distribution. When we schedule a job on an instance, we can also specify a budget cap. For a spot instance i, let $\pi_{i,s}$ be the probability that it uses exactly s dollars before it gets interrupted. For a spot instance i and job j, let $R_{(j,i)}(s)$ denotes the progress it achieves before the last check point, e.g. number of trained epochs. Notice the function $R_{(j,i)}(\cdot)$ is monotone, i.e. $R_{(j,i)}(s) \leq R_{(j,i)}(s')$ if $s \leq s'$. When we schedule job j onto instance i, some processing time is wasted on environment setup and checkpoint restoration, which does not count towards progress. This is captured by setting $R_{(j,i)}(s) = 0$ if s dollars is not enough to finish the first epoch. An on-demand instance and a spot instance of the same configuration are considered different instances. The utility of a job is a submodular function (more details later), which captures diminishing returns of ML training. With a given budget B, we would like to maximize the total expected utility of all jobs.

In this work, we focus on the offline case where price and interruption distribution never changes. To deal with the online updates in parameters of instances or a refreshed quota, we can simply re-run an algorithm for this case on the updated information to find a new schedule.

4.2.3 Final Problem Statement

This problem is further mapped into the following correlated stochastic knapsack problem that maximizes a set submodular function. There are n items, each item takes a random size $\mathsf{size}_i \in \mathbb{N}$ with probability $p_i(s)$, and gets a reward $r_i \in [M]$. Each size corresponds to reward. In other words, for each item i, there is a reward function $R_i : [\mathbb{N}] \to [M]$, such that $r_i = R_i(\mathsf{size}_i)$. We assume R_i to be non-decreasing, i.e., the larger an item, the more reward it deserves. We further require that the chosen set of items S be an independent set of a partition matroid[4] $\mathcal{I} = \{\mathcal{I}_k\}_{k\in[K]}$. We have a budget $B \in \mathbb{N}$ for the total size of items, and wish to extract as much reward as possible. The total reward is a lattice-submodular function [5] $f : [M]^n \to \mathbb{R}^+$ on the rewards of every item. Let $S \subseteq [n]$, we sample a vector $q \in [M]^n$ as follows. Each component $q(i)$ is sampled independently. For $i \in S$, $\Pr[r_i = R_i(s)] = p_i(s)$; for $i \notin S$, $r_i = 0$ with probability 1. Denote this distribution as q_S. Then the objective is to find a (random) set $S \subseteq \mathcal{I}$ of items that maximizes $\mathbb{E}_{\theta\sim q_S}[f(\theta)]$ subject to $\sum_{i\in S} \mathsf{size}_i \le B$.

The reduction is as follows. We consider instances with quota larger than one

[4]for a partition matriod $\mathcal{I} = \{\mathcal{I}_k\}_{k\in[K]}$, set S is considered independent iff $\forall k, S \cap \mathcal{I}_k \le 1$
[5]See definition in Section 4.2.2

as multiple independent copies. For each job j, instance i and budget cap b, we define an item (j,i,b), where $p_{(j,i,b)}(s) = \pi_{i,s}$ when $s < b$; $p_{(j,i,b)}(s) = \sum_{s' \geq s} \pi_{i,s'}$ when $s = b$; and 0 otherwise. The new reward function is exactly $R_{(j,i,b)}(\cdot)$. Notice for each instance, only a single job can be scheduled on it, and a specific budget cap can be chosen, we further impose a partition matriod $\{\mathcal{I}_i\}_{i \in [K]}$ on the items, where $\mathcal{I}_i = \{(j,i,b) | \forall j, \forall b\}$.

We consider adaptive policies, i.e. we can choose an item to include, observe its realized size, and make further decisions based on the feedback. At first, only the size distribution of items are known. When the policy includes an item, its size size_i is realized, together with its reward r_i. An adaptive policy can make its decision based on all the realizations it has seen so far. An non-adaptive policy, on the other hand, does not see the realizations. All it can do is propose a S of items, and hope for the best. In this work, we only consider adaptive policies without cancellation, i.e., the inclusion of an item is irrevocable.

For a vector $q \in [M]^n$, let $\Pr_\pi[q]$ denote the probability that we get outcome q when running policy π. Note this probability is with respect to the randomness in the state of items and in the the policy π. Let $f_{\mathsf{avg}}(\pi)$ denote $\sum_{q \in [B]^n} \Pr[q] f(r)$, i.e., the average objective value obtained by π. Our aim is to find a policy π that maximizes $f_{\mathsf{avg}}(\pi)$. We say π is an α-approximation policy if $f_{\mathsf{avg}}(\pi) \geq \alpha f_{\mathsf{avg}}(\pi^*)$ for any policy π^*.

4.2.4 Eliminating an Assumption

In Fukunaga et al. [28], the authors considered the stochastic knapsack problem with a lattice-submodular target function. They made two assumptions. The first assumption states that larger size means larger reward for every particular job. This is a reasonable assumption for our spot instance scheduling problem, and remains crucial in the analysis. We eliminate the need of the second assumption. This assumption states that we will never select an item which could overflow the budget, given the realization of selected items. For example, suppose we are left with a remaining budget of 20 at some time, and all items have a 0.001 probability of size 21. What this assumption suggests is that none of the items is allowed to be selected. However, for many cases, selecting such an item is a desirable choice since additional value is obtained with high probability. If we are unlucky and the size goes beyond the remaining budget, we either receive a partial value, or do not get any value at all. We achieve the elimination via a budget cap b, which creates the truncated version of item i at budget b. Item (i, b) is only available to be scheduled when the remaining budget is $B - b$. Depending on whether we allow partial reward, we set the distribution of size and rewards for item (i, b). If we get 0 reward when the item overflows, then $p_{(i,b)}(s) = p_i(s)$ when $0 < s \leq b$, and $p_{(i,b)}(s) = 0$ otherwise. If we allow the item to collect a reward for size τ if overflow happens, then $p_{(i,b)}(s) = p_i(s)$ when $0 \leq s < b$, and $p_{(i,b)}(b) = 1 - \sum_{s<\tau} p_i(s)$. It is not hard to see that these new items are valid, and we will see later that the limitation on which slot to schedule can be handled easily via a time indexed LP. Another thing to notice is that at most

one of the items in $\{i\} \cup \{(i,b)|b \in [B]\}$ can be selected, which is captured by the partition matroid constraint.

4.3 Continuous Optimization Phase

Like most submodular maximization problems, our algorithms consists of two phases, a continuous optimization phase and a rounding phase. In this section, we describe the continuous optimization phase. Given a lattice-submodular function $f : [\mathbb{M}]^n \to \mathbb{R}^+$, we define a submodular set-function $\bar{f} : 2^n \to \mathbb{R}_+$. Then $\bar{f} := \mathbb{E}_{r \sim q_S}[f(r)]$ for any $S \subseteq [n]$. If f is lattice-submodular, then $\bar{f}$ is guaranteed to be a monotone set-submodular function [71]. We define $\bar{F} : 2^n \to \mathbb{R}^+$ to be the multilinear extension of $\bar{f}$, i.e., $F(y) = \sum_{S \subseteq [n]} \prod_{i \in S} y_i \prod_{i' \notin S}(1 - y_{i'})\bar{f}(S)$. Note evaluating the function F can take exponential time, but it can be approximated within a multiplicative factor of $(1+\epsilon)$ for any constant $\epsilon > 0$, which is standard in submodular maximization, see [73]. In this paper, we assume it can be evaluated exactly for simplicity. If $\bar{w}(i)$ is the probability that item i is in S, then the target function we are maximizing would be $\bar{F}(\bar{w})$. What remains are the constraints.

4.3.1 Stochastic Knapsack Exponential Constraints

Consider a certain item i, we replace it with an equivalent Markovian bandit. It starts at state ρ_i. The first pull will move it to one of the states $u_i(1,*)$, arriving at $u_i(1,s)$ with probability $p_{\rho_i,u_i(1,s)} = p_i(s)$ (the corresponding item has size s). When this arm is pulled, we are forced to keep pulling the same arm until arriving at the

termination state $\emptyset_i$. A state $u_i(k,s)$ indicates that this arm has used up k units of time, and have arrived at state $u_i(1,s)$ (in other words, the corresponding item have size s). Therefore, if $k < s$, it will transit to state $u_i(k+1,s)$ with probability $p_{u_i(k,s),u_i(k+1,s)} = 1$. Otherwise, with probability $p_{u_i(k,s),\emptyset_i} = 1$, transit to state $\emptyset_i$. We reform the constraints in Ma [70] as follows. Let $S_i = \{u_i^{*,*}\} \cup \{\emptyset_i\}$ for all $i \in [n]$. Let $\boldsymbol{\mathcal{S}}' = \{\pi : \pi_i \notin \{\rho_i, \emptyset_i\}, \pi_j \notin \{\rho_j, \emptyset_j\}, i \neq j\}$, the set of states where at least two arms are in the middle of processing at the same time. Let $\boldsymbol{\mathcal{S}}'' = \{\pi : \pi_i \neq \rho_i \text{ and } \pi_j \neq \rho_j, i, j \in \mathcal{I}_k \text{ for some } k\}$, the set of states where some conflicting arms (due to the partition matriod) have been started. Define $\boldsymbol{\mathcal{S}} := S_1 \times \cdots \times S_n \setminus (\boldsymbol{\mathcal{S}}' \cup \boldsymbol{\mathcal{S}}'')$, which is the set of all valid states. Let $I(\pi) = \{i : \pi_i \neq \emptyset_i\}$ be the set of arms that could be played from π. Let π^u denote the joint node where the i-th component is replaced by u. Let $y_{\pi,t}$ be the probability of at state π at time t, and $z_{\pi,i,t}$ be the probability that we pull arm i at time t, when the current state is π.

$$\sum_{i \in I(\pi)} z_{\pi,i,t} \leq y_{\pi,t} \qquad \pi \in \boldsymbol{\mathcal{S}}, t \in [B] \tag{4.1}$$

$$z_{\pi,i,t} = y_{\pi,t} \qquad \pi \in \boldsymbol{\mathcal{S}}, i : \pi_i \in S_i \setminus \{\rho_i, \emptyset_i\}, t \in [B] \tag{4.2}$$

$$z_{\pi,i,t} \geq 0 \qquad \pi \in \boldsymbol{\mathcal{S}}, i \in [n], t \in [B] \tag{4.3}$$

Let $\boldsymbol{\mathcal{A}}_i = \{\pi \in \boldsymbol{\mathcal{S}} : \pi_i \notin \{\rho_i, \emptyset_i\}\}$, the joint node with arm i in the middle of processing. We call arm i the *active* arm. Let $\boldsymbol{\mathcal{A}} = \bigcup_{i=1}^n \boldsymbol{\mathcal{A}}_i$, the set of all states where some arm is in the middle of processing. For state $\pi \in \boldsymbol{\mathcal{S}}$, let $\boldsymbol{\mathcal{P}}(\pi)$ denote the subset of $\boldsymbol{\mathcal{S}}$ that would transit to π with no play: if $\pi \notin \boldsymbol{\mathcal{A}}$, then

$\boldsymbol{\mathcal{P}}(\pi) = \{\pi\} \cup (\bigcup_{i \notin I(\pi)} \{\pi^u : u \in \mathcal{S}_i \setminus \{\rho_i\}\}$; if $\pi \in \boldsymbol{\mathcal{A}}$, then $\boldsymbol{\mathcal{P}}(\pi) = \emptyset$. Define $\mathsf{Par}(\pi) = \{v \in S : p_{v,u} > 0\}$, the node that have a positive probability of transitioning to u. Then y-variables are updated as follows:

$$y_{(\rho_1,\ldots,\rho_n),1} = 1 \tag{4.4}$$

$$y_{\pi,1} = 0 \qquad \pi \in \boldsymbol{\mathcal{S}} \setminus \{(\rho_1, \cdots, \rho_n)\} \tag{4.5}$$

$$y_{\pi,t} = \sum_{\pi' \in \boldsymbol{\mathcal{P}}(\pi)} \left(y_{\pi',t-1} - \sum_{i \in I(\pi')}^{n} z_{\pi',i,t-1} \right) \qquad t > 1, \pi \in \boldsymbol{\mathcal{S}} \setminus \boldsymbol{\mathcal{A}} \tag{4.6}$$

$$y_{\pi,t} = \sum_{\rho_i \in \mathsf{Par}(\pi_i)} \left(\sum_{\pi' \in \boldsymbol{\mathcal{P}}(\pi^{\rho_i})}^{n} z_{\pi',i,t-1} \right) \cdot p_{\rho_i,\pi_i} \quad t > 1, i \in [n], \pi \in \boldsymbol{\mathcal{A}}_i, \pi_i \in \{u_i^{1,*}\} \tag{4.7}$$

$$y_{\pi,t} = \sum_{u \in \mathsf{Par}(\pi_i)} z_{\pi^u,i,t-1} \cdot p_{u,\pi_i} \qquad t > 1, i \in [n], \pi \in \boldsymbol{\mathcal{A}}_i, \pi_i \notin \{u_i^{1,*}\} \tag{4.8}$$

Equation (4.6) updates $y_{\pi,t}$ for $\pi \notin \boldsymbol{\mathcal{A}}$, i.e. joint nodes with no active arms. Such a joint node π can only come from a no-play from a joint node in $\boldsymbol{\mathcal{P}}(\pi)$. Equation (4.7), Equation (4.8) update $y_{\pi,t}$ for $\pi \in \boldsymbol{\mathcal{A}}$. To get to the joint node π, we must have played arm i in previous step(s). In Equation (4.7), we consider the case if π_i is one of $u_i(1,*)$. We were at ρ_i right before, so it is possible that in the last step, we switched to π^{ρ_i} from some joint node in $\boldsymbol{\mathcal{P}}(\pi^{\rho_i})$ without playing an arm. In Equation (4.8), we consider other cases, in which case arm i was played at time $t-1$. These equations guarantee that at each time step, $y_{*,t}$ form a distribution, i.e. $\sum_{\pi \in \boldsymbol{\mathcal{S}}} y_{\pi,t} = 1$. Combined with Equation (4.1), we get

$$\sum_{\pi \in S} \sum_{i \in I(\pi)} z_{\pi,i,t} \leq 1 \qquad t \in [B].$$

The exponential constraints are a combination of Equation (4.1), (4.2), (4.3), (4.4), (4.5), (4.6), (4.7), (4.8).

4.3.2 Stochastic Knapsack Polynomial Constraints

The previous formulation is exponential due to the size of $\boldsymbol{\mathcal{S}}$. In order to solve in polynomial time, we relax it by no longer consider the joint distribution of items. Let $s_{u,t}$ be the probability that arm i is on node u at the beginning of time t. Let $x_{u,t}$ be the probability that we pull an arm on node u at time t. The objective would persist, with $\bar{x}$, where $\bar{x}(u) = \sum_t x_{u,t}$. The new set of constraints is

$$x_{u,t} \leq s_{u,t} \qquad u \in \mathcal{S}, t \in [B] \tag{4.9}$$

$$x_{u,t} = s_{u,t} \qquad u \in \bigcup_{i \in [n]} \{u^{*,*}\}, t \in [B] \tag{4.10}$$

$$x_{u,t} \geq 0 \qquad u \in \mathcal{S}, t \in [B] \tag{4.11}$$

$$\sum_{u \in \mathcal{S}} x_{u,t} \leq 1 \qquad t \in [B] \tag{4.12}$$

There is a constraint due to the partition matriod of arms (recall $\mathcal{I}_k$ is one partition of the partition matroid).

$$\sum_{i \in \mathcal{I}_k} s_{\rho_i,1} = 1, \qquad \forall \mathcal{I}_k \tag{4.13}$$

$$s_{\rho_i,1} \geq 0, \qquad i \in [n] \tag{4.14}$$

And here are the state transition constraints.

$$s_{u,1} = 0 \qquad u \in \mathcal{S} \setminus \{\rho_1, \cdots, \rho_n\} \tag{4.15}$$

$$s_{\rho_i,t} = s_{\rho_i,t-1} - x_{\rho_i,t-1} \qquad t > 1, i \in [n] \tag{4.16}$$

$$s_{u,t} = \sum_{v \in \mathsf{Par}(u)} x_{v,t-1} \cdot p_{v,u} \qquad t > 1, u \in \mathcal{S} \setminus \{\rho_{1,\ldots,\rho_n}\} \tag{4.17}$$

We denote this polynomial program with PolyP. For any program P, let $\mathsf{OPT}_{\mathrm{P}}$ denote its optimal value. We state without proof for the following theorem.

Theorem 5 (reformation of Lemma 2.3 from Ma [70])**.** *Given a feasible solution* $\{z_{\pi,i,t}\}, \{y_{\pi,t}\}$ *to ExpP, we can construct a solution to PolyP with the same objective value by setting* $x_{u,t} = \sum_{\pi \in \boldsymbol{\mathcal{S}}:\pi_i=u} z_{\pi,i,t}$, $s_{u,t} = \sum_{\pi \in \boldsymbol{\mathcal{S}}:\pi_{i=u}} y_{\pi,t}$ *for all* $i \in [n]$, $u \in [0,1]$, $t \in B$. *Thus, the feasible region of PolyP is a projection of that of ExpP onto a subspace and* $\mathsf{OPT}_{\mathsf{ExpP}} \leq \mathsf{OPT}_{\mathsf{PolyP}}$.

4.3.3 Construct a solution $\{z_{\pi,i,t}, y_{\pi,t}\}$ of ExpP from a solution $\{x_{u,t}, s_{u,t}\}$

Our objective is that the new solution will obtain half the objective value of PolyP. It will satisfy

$$\sum_{\pi \in \boldsymbol{\mathcal{S}}:\pi_i=u} z_{\pi,i,t} = \frac{x_{u,t}}{2} \quad i \in [n], u \in \mathcal{S}_i, t \in [B].$$

We define specific $\{z_{\pi,i,t}, y_{\pi,t}\}$ over B iterations $t = 1, \ldots, B$. On iteration t:

- Compute $y_{\pi,t}$ for all $\pi \in \boldsymbol{\mathcal{S}}$.

- Define $\tilde{y}_{\pi,t} = y_{\pi,t}$ if $\pi \notin \boldsymbol{\mathcal{A}}$, and $\tilde{y}_{\pi,t} = y_{\pi,t} - \sum_{a \in A} z_{\pi,i,t}$ if $\pi \in \boldsymbol{\mathcal{A}}_i$ for some $i \in [n]$ (if $\pi \in \boldsymbol{\mathcal{A}}_i$, then $\{z_{\pi,i,t} : a \in A\}$ has already been set in a previous iteration).

- For all $i \in [n]$, define $f_{i,t} = \sum_{\pi \in \boldsymbol{\mathcal{S}} : \pi_i = \rho_i} \tilde{y}_{\pi,t}$.

- For all $i \in [n]$, $\pi \in \boldsymbol{\mathcal{S}}$ such that $\pi_i = \rho_i$, and $a \in A$, set $z^a_{\pi,i,t} = \tilde{y}_{\pi,t} \cdot \frac{1}{2} \cdot \frac{x_{\rho_{i,t}}}{f_{i,t}}$.

- For all $i \in [n]$, $\pi \in \boldsymbol{\mathcal{S}}$ such that $\pi_i = \rho_i$ and $\pi_j \in \{\rho_j, \phi_j\}$ for $j \neq i$, define $g_{\pi,i,t} = \sum_{\pi' \in \mathcal{P}(\pi)} z_{\pi',i,t}$.

- For all $i \in [n]$, $u \in \mathcal{S}_i \setminus \{\rho_i\}$, $\pi \in \boldsymbol{\mathcal{S}}$ such that $\pi_i = u$, and $a \in A$, set $z^a_{\pi,i,t+\mathsf{depth}(u)} = g_{\pi^{\rho_i},i,t} \cdot \frac{x^a_{u,t+\mathsf{depth}(u))}}{x_{\rho_i,t}}$.

4.3.4 Continuous Optimization

In order to solve PolyP, we use the Stochastic Continuous Greedy algorithm. This algorithm maximizes the multilinear extension G of a monotone set-submodular function g over a solvable downward-closed polytope. A polytope $\mathcal{P} \subseteq [0,1]^{\mathcal{N}}$ is considered *solvable* if we can find an algorithm to optimize linear functions over it, and downward-closed if $x \in \mathcal{P}$ and $0 \leq y \leq x$ imply $y \in \mathcal{P}$. In our case, $\mathcal{P}$ is solvable due to its linearity, and that solving a linear program falls in **P**. Note $\mathcal{P}$ is down-monotone. The algorithm involves a controlling parameter called *stopping time*. For a stopping time $0 < b \leq 1$, the algorithm outputs a solution x such that $x/b \in \mathcal{P}$, while $G(x) \geq (1 - e^{-b} - O(n^3\delta)) \max_{y \in Q} G(y)$, where n is the size of the

set over which g is defined and δ is the step size used in the algorithm. Here $\mathcal{P}$ is assumed to include the characteristic vector of every singleton set.

Theorem 6 (reformation of Theorem 3 from Fukunaga et al. [28]). *If the stochastic continuous greedy algorithm with stopping time $b = 1/2 \in (0, 1]$ and step size $\delta = o(|I|^{-3})$ is applied to program* PolyP*, then the algorithm outputs a solution $x \in b\mathcal{P}$ such that $\bar{F}(\bar{x}) \geq (1 - e^{-b} - o(1)) f_{\mathsf{avg}}(\pi^*)$ for any adaptive policy π^*.*

4.4 Rounding Phase

Now that we have a fractional solution x, we proceed to round it to an integral policy. We need a variant of the contention resolution scheme that was introduced as a general framework for designing rounding algorithms that maximizes expected submodular functions ([72, 74, 75]). The variant is an extension from a set submodular function to a lattice-submodular function, first introduced in Fukunaga et al. [28].

4.4.1 Contention Resolution Scheme

Let $f : [B]^n \to \mathbb{R}_+$ be a monotone lattice-submodular function and the probability distribution $q_i : [B] \to [0, 1]$ on $[B]$ be given for each $i \in I$. We write $v \sim q$ if $v \in [B]^n$ is a random vector such that, for each $i \in [n]$, the corresponding component $v(i)$ is determined independently as $j \in [B]$ with probability $q_i(j)$. Let $\mathcal{F} \subseteq [B]^n$ be a downward-closed subset of $[B]^n$ (i.e., if $u \leq v \in \mathcal{F}$, then $u \in \mathcal{F}$), and let $\alpha \in [0, 1]$. We have the following definition for an α-contention resolution scheme (α-CRS).

Definition 1 (α-Contention Resolution Scheme (α-CRS))**.** *A mapping* $\psi : [B]^n \to \mathcal{F}$ *is an* α*-Contention Resolution Scheme with respect to* q *if it satisfies:*

1. $\psi(v)(i) \in \{v(i), 0\}$ *for each* $i \in [n]$*;*

2. *if* $v \sim q$*, then* $\Pr[\psi(v)(i) = j | v(i) = j] \geq \alpha$ *holds for all* $i \in I$ *and* $j \in B$*. The probability is based on randomness both in* v *and in* ψ *when* ψ *is randomized.*

Definition 2 (monotone α-CRS)**.** *An* α*-CRS* ψ *is considered* monotone*, if, for each* $u, v \in [B]^n$ *such that* $u(i) = v(i)$ *and* $u \leq v$*,* $\Pr[\psi(u)(i) = u(i)] \geq \Pr[\psi(v)(i) = v(i)]$ *holds. The probability is based only on the randomness of* ψ*.*

We have the following lemma:

Lemma 5 (Theorem 4 from Fukunaga et al. [28])**.** *If* ψ *is a monotone* α*-CRS with respect to* q*, then* $\mathbb{E}_{v \sim q}[f(\psi(v))] \geq \alpha \mathbb{E}_{v \in q}[f(v)]$.

4.4.2 Rounding Algorithm

For each item i, we propose to include it at time t with probability $x_{\rho_i,t}$, and drop it with probability $1 - \sum_t x_{\rho_i,t}$. Now we have a set $R' = \{(i,t)\}$ of proposed item time pairs. We sort the set according to t, and include the items one by one. For a pair (i,t), we will include item i if time t is available. After including it in our solution, we get its realized size, and mark the corresponding time slots unavailable. If it is not available, we will simulate its inclusion, and sample its size size_i should it be included. We also mark those time slots unavailable even though this item is not included. This seemingly wasteful step is to ensure that the rounding scheme is monotone.

Algorithm 4: Rounding Algorithm

```
1  for Partition group I_k do
2      Sample (i,t) from I_k × [C], get (i,t) with probability x_{ρ_i,t}, and gets ∅
         with probability 1 − ∑_{i∈I_k} ∑_t x_{ρ_i,t};
3      if not get ∅ then
4          I ← I ∪ {(i,t)} ;
5      end
6  end
7  Sort I according to a non-decreasing ordering of t, break ties uniformly at
     random ;
8  C = 0, S = ∅, mark all times slots available ;
9  for (i,t) ∈ I do
10     if time slot t is available then
11         Include item i ;
12         Observe s_i ;
13     else
14         Simulate including item i, and observe s_i;
15     end
16     Mark time slots from t to t + s_i unavailable;
17 end
```

First note $\sum_{i\in\mathcal{I}_k}\sum_t x_{\rho_i,t} \leq 1$ for all partition $\mathcal{I}_k$. We have $\sum_t x_{\rho_i,t} \leq s_{\rho_i,1}$ due to Equation (4.16), which lead to our claim with the help of Equation (4.13). Therefore, our algorithm is well-defined. The remaining of this section is devoted to proving the following theorem.

Theorem 7. *Let π denote Algorithm 4, and x denote the solution we get from PolyP. Then $f_{avg}(\pi) \geq \bar{F}(\bar{x})/2$.*

We define two mappings $\sigma(\cdot)$ and $\omega(\cdot)$, where the first (roughly) corresponds to the step that maps x to I in Algorithm 4, and $\omega(\cdot)$ corresponds to the mapping from set I to the final output. The mapping $\sigma(v)$ receives a vector $\bar{x} \in [0,1]^n$ and returns a random vector $v \in [B]^n$. From each partition $\mathcal{I}_k$, we pick at most one i, each $i \in \mathcal{I}_k$ is picked with probability $\sum_t x_{\rho_i,t}$. If it is picked, the i-th component $v(i)$

independently takes value j with probability $p_i(j)$, and 0 otherwise, which happens with probability $1 - \sum_j p_i(j)$. This captures the construction of set I (only the item part, note $\Pr[\sigma(x)(i) > 0] = \Pr[\exists t, \text{s.t.} (i,t) \in I]$), together with the random outcome of the item. The mapping $\omega(\cdot)$ maps $v \in [B]^n$ to $w \in [B]^n$. To mimic Algorithm 4, we first assign time value $t(i)$ to each component $v(i)$, according to $x_{\rho_i,t}$. Based on $t(i)$, we form a precedence ordering $\prec$ between i after random tie breaking (a random tie breaking is crucial). Then, we set $\omega(v)(i) = 0$ if there exists a component $j \prec i$ such that $t(j) \leq t(i) < t(j) + v(i)$, and $w(v)(i) = v(i)$ otherwise. We can observe that given input x, Algorithm 4 outputs exactly $\omega(\sigma(x))$ if the random realized sizes of items are the same.

We first prove the following helping lemma.

Lemma 6. *Let $X_1, \ldots, X_n$ be $\{0,1\}$ random variables with $E[X_i] = x_i$. Suppose X_i form a partition matroid $\mathcal{I}$ with partition $\mathcal{I}_1, \ldots, \mathcal{I}_K$, and f is a submodular set function on $[n]$ with $f(\emptyset) = 0$, then $\mathbb{E}[f(X_1, \ldots, X_n)] \geq \sum_{S \subseteq [n]} \prod_{i \in S} x_i \prod_{i' \notin S}(1 - x_{i'}) f(S)$.*

Proof of Lemma 6. We prove by induction on K, the number of partitions. When $K = 1$, at most one X_i is one. Let $\mathbf{0}$ denote the all 0 vector, and $\mathbf{1}_i$ denote the vector where the i-th entry is 1 while all other entries are zero. Therefore,

$$\begin{aligned}
&\mathbb{E}[f(X_1, \ldots, X_n)] \\
=&\Pr[\forall i, X_i = 0] f(\emptyset) + \sum_{i=1}^{n} \Pr[X_i = 1] f(\{i\}) \\
=&\sum_{i=1}^{n} x_i f(\{i\}).
\end{aligned}$$

On the other hand,

$$
\begin{aligned}
&\sum_{S\subseteq[n]}\prod_{i\in S}x_i\prod_{i'\notin S}(1-x_{i'})f(S)\\
\leq&\sum_{S\subseteq[n]}\prod_{i\in S}x_i\prod_{i'\notin S}(1-x_{i'})(\sum_{i''\in S}f(\{i''\}))\\
=&\sum_{i''}f(\{i''\})\sum_{\substack{S\subseteq[n]\\ i''\in S}}\prod_{i\in S}x_i\prod_{i'\notin S}(1-x_{i'})\\
=&\sum_{i''}x_{i''}f(\{i''\})\sum_{S\subseteq[n]\setminus\{i\}}\prod_{i\in S}x_i\prod_{i'\notin S}(1-x_{i'})\\
=&\sum_{i''}x_{i''}f(\{i''\})\\
=&\mathbb{E}[f(X_1,\ldots,X_n)]
\end{aligned}
$$

Suppose the lemma is true for $K=K^0$. Let $f_S(X_1,\ldots,X_n)$ denote the function f where $X_i=0,\forall i\notin S$. We slightly abuse the notations and use $f(S)$ to denote $f_S(X_1,\ldots,X_n)$ when there is no confusion. On the one hand, we have

$$
\begin{aligned}
\mathbb{E}[f(X_1,\ldots,X_n)]=&(1-\sum_{i\in\mathcal{I}_1}\Pr[X_i=1])f_{\sigma_k=0}(X_1,\ldots,X_n)\\
&+\sum_{i\in\mathcal{I}_1}\Pr[X_i=1]\mathbb{E}[f_{\sigma_k=i}(X_1,\ldots,X_n)].
\end{aligned}
$$

On the other hand,

$$
\begin{aligned}
&\sum_{S\subseteq[n]}\prod_{i\in S}x_i\prod_{i'\notin S}(1-x_{i'})f(S)\\
\leq&\sum_{S\subseteq[n]}\prod_{i\in S}x_i\prod_{i'\notin S}(1-x_{i'})\left(f(S\setminus\mathcal{I}_1)+\sum_{i\in\mathcal{I}_1\cap S}(f(\{i\}\cup(S\setminus\mathcal{I}_1))-f(S\setminus\mathcal{I}_1))\right)
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_{S\subseteq[n]} \prod_{i\in S} x_i \prod_{i'\notin S}(1-x_{i'}) f(S\setminus \mathcal{I}_1) \\
&\quad + \sum_{S\subseteq[n]} \prod_{i\in S} x_i \prod_{i'\notin S}(1-x_{i'}) \sum_{i\in \mathcal{I}_1\cap S} (f(\{i\}\cup(S\setminus\mathcal{I}_1)) - f(S\setminus\mathcal{I}_1)) \\
&= \sum_{S\subseteq[n]} \prod_{\substack{i'\in S\\ i'\in\mathcal{I}_1\setminus\{i\}}} x_{i'} \prod_{\substack{i''\notin S\\ i''\in\mathcal{I}_1\setminus\{i\}}} (1-x_{i''}) \prod_{\substack{j\in S\\ j\notin\mathcal{I}_1}} x_i \prod_{\substack{j'\notin S\\ j\notin\mathcal{I}_1}} (1-x_{i'})(1-x_{i'}) f(S) \\
&\quad + \sum_{i\in\mathcal{I}_1} x_i \sum_{\substack{S\subseteq[n]\\ S\ni i}} \prod_{\substack{i'\in S\\ i'\in\mathcal{I}_1\setminus\{i\}}} x_{i'} \prod_{\substack{i''\notin S\\ i''\in\mathcal{I}_1\setminus\{i\}}} (1-x_{i''}) \prod_{\substack{j\in S\\ j\notin\mathcal{I}_1}} x_i \prod_{\substack{j'\notin S\\ j\notin\mathcal{I}_1}} (1-x_{i'})(f(\{i\}\cup(S\setminus\mathcal{I}_1)) - f(S\setminus\mathcal{I}_1)) \\
&= \sum_{S\subseteq[n]\setminus\mathcal{I}_1} \prod_{i\in S} x_i \prod_{i'\notin S}(1-x_{i'}) f(S) \\
&\quad + \sum_{i\in\mathcal{I}_1} x_i \sum_{\substack{S\subseteq[n]\setminus\mathcal{I}_1\\ S\ni i}} \prod_{\substack{j\in S\\ j\notin\mathcal{I}_1}} x_i \prod_{\substack{j'\notin S\\ j\notin\mathcal{I}_1}} (1-x_{i'})(f(\{i\}\cup(S\setminus\mathcal{I}_1)) - f(S\setminus\mathcal{I}_1)) \\
&= \sum_{S\subseteq[n]\setminus\mathcal{I}_1} \prod_{i\in S} x_i \prod_{i'\notin S}(1-x_{i'}) f(S) \\
&\quad + \sum_{i\in\mathcal{I}_1} x_i \sum_{S\subseteq[n]\setminus\mathcal{I}_1} \prod_{\substack{j\in S\\ j\notin\mathcal{I}_1}} x_i \prod_{\substack{j'\notin S\\ j\notin\mathcal{I}_1}} (1-x_{i'})(f(\{i\}\cup(S)) - f(S)) \\
&=(1-\sum_{i\in\mathcal{I}_1} x_i) \sum_{S\subseteq[n]\setminus\mathcal{I}_1} \prod_{i\in S} x_i \prod_{i'\notin S}(1-x_{i'}) f(S) \\
&\quad + \sum_{i\in\mathcal{I}_1} x_i \sum_{S\subseteq[n]\setminus\mathcal{I}_1} \prod_{\substack{j\in S\\ j\notin\mathcal{I}_1}} x_i \prod_{\substack{j'\notin S\\ j\notin\mathcal{I}_1}} (1-x_{i'}) f(\{i\}\cup(S)) \\
&\geq(1-\sum_{i\in\mathcal{I}_1}\Pr[X_i=1]) f_{[n]\setminus\mathcal{I}_1}(X_1,\ldots,X_n) + \sum_{i\in\mathcal{I}_1}\Pr[X_i=1]\mathbb{E}[f_{[n]\setminus(\mathcal{I}_1\setminus\{i\})}(X_1,\ldots,X_n)] \\
&=\mathbb{E}[f(X_1,\ldots,X_n)]
\end{aligned}
$$

The first inequality is the submodular property of f. The last inequality applies the induction hypothesis. $\square$

Now we are ready to bound $\mathbb{E}[f(\sigma(x))]$ with the following lemma.

Lemma 7. *$\mathbb{E}[f(\sigma(x))] \geq \bar{F}(\bar{x})$ holds for any $x \in P$.*

Proof of Lemma 7. Consider the distribution of $\sigma(x)$. Let $X_{i,j}$ denote the event that item i has size j. We know $\Pr[X_{i,j}] = \bar{x} \cdot p_i(j)$. Suppose the partition matriod $\mathcal{I}$ has partition $\mathcal{I}_1, \ldots, \mathcal{I}_K$, then the set $\{(i,j) | i \in [n], j \in [M]\}$ forms a partition matriod $\mathcal{I}'$, where $\mathcal{I}'_k = \{(i,j) | i \in \mathcal{I}_k, j \in [M]\}$. Applying Lemma 6, we get $\mathbb{E}[f(\sigma(x))] \geq \bar{F}(x_1, \ldots, x_n) = \bar{F}(\bar{x})$. □

With Lemma 6, we are ready to prove

Lemma 8. *ω is a monotone $1/2$-CRS with respect to $\bar{x}$.*

Proof of Lemma 8. We first prove ω is a 1/2-CRS, then prove it monotone. The first condition is obvious due to the definition of $w(\cdot)$. The second condition is asking a proof for $\Pr[w(v)(i) = j | v(i) = j] \geq 1/2$. In the language of the rounding algorithm, let Drop_i denotes the event (respect to the randomness in ω and v) that we drop the pair (i,t). It is the same as proving $\Pr[\mathsf{Drop}_i | \text{item } i \text{ is selected at time } t] \leq \frac{1}{2}$.

Due to the way we round the solution, item i is only included once, so item i is always available. Conditioned on the event that item $i \in I$, the only reason that it is dropped is when some item j ($j \prec i$) marked the time slot t unavailable. Fix j, the probability that it marked time slot t unavailable is

$$\begin{aligned}
&\frac{1}{2}\sum_{t'=1}^{t-1} x_{\rho_j,t'} \cdot \Pr[\mathsf{size}_j \geq t - t'] + \Pr[\text{item } j \text{ is considered before } i] \cdot \frac{1}{2} x_{\rho_j,t} \\
\leq &\frac{1}{2}\sum_{t'=1}^{t-1} x_{\rho_j,t'} \cdot \Pr[\mathsf{size}_j \geq t - t'] + \frac{1}{2} x_{\rho_j,t}.
\end{aligned}$$

Where

$$
\begin{aligned}
&\sum_{t'=1}^{t-1} x_{\rho_j,t'} \cdot \Pr[\mathsf{size}_j \geq t - t'] \\
=&\sum_{t'=1}^{t-1}\sum_{\tau=t-t'}^{B-t} x_{\rho_j,t'} \Pr[\mathsf{size}_j = \tau] \\
=&\sum_{t'=1}^{t-1}\sum_{\tau=t-t'}^{B-t} x_{u_j(1,\tau),t'+1} \\
=&\sum_{t'=1}^{t-1}\sum_{\tau=t-t'}^{B-t} x_{u_j(t-t',\tau),t} \\
\leq& \sum_{u\in\{\emptyset_j\}\cup\{u_i(*,*)\}} x_{u,t}
\end{aligned}
$$

Therefore, the total probability that item i is blocked by any item is upper bounded by union bound:

$$
\begin{aligned}
&\frac{1}{2}\sum_{j\neq i}\sum_{t'=1}^{t} x_{\rho_j,t'} \cdot \Pr[\mathsf{size}_j \geq t - t'] + \frac{1}{2}\sum_{j\neq i} x_{\rho_j,t} \\
\leq&\frac{1}{2}\sum_{j\neq i}\sum_{u\in\{\emptyset_j\}\cup\{u_i(*,*)\}} x_{u,t} + \frac{1}{2}\sum_{j\in[n]} x_{\rho_j,t} \\
\leq&\frac{1}{2}\sum_{j\in[n]}\sum_{u\in\{\emptyset_j\}\cup\{u_i(*,*)\}} x_{u,t} + \frac{1}{2}\sum_{j\in[n]} x_{\rho_j,t} \\
\leq&\frac{1}{2}(1-\sum_{j\in n} x_{\rho_j,t}) + \frac{1}{2}\sum_{j\in[n]} x_{\rho_j,t} \\
=&\frac{1}{2}.
\end{aligned}
$$

Lastly, we show ω is monotone. Suppose vectors $u, v \in [B]^n$ satisfies $u \leq v$, and $u(i) = v(i) = j > 0$. We only need to show $\Pr[\omega(u)(i) = j] \geq \Pr[\omega(v)(i) = j]$, where

randomness is with respect to the choice of time and ordering. In this case,

$$\Pr[w(u)(i) = j] = \sum_{t=1}^{B} x_{\rho_i,t} \left(1 - \sum_{i'} \Pr[i' \prec i | t(i) = t] \sum_{t'=t-u(i')}^{t} x_{\rho_{i'},t'}\right).$$

Therefore

$$\begin{aligned}
&\Pr[w(u)(i) = j] - \Pr[w(v)(i) = j] \\
=&\sum_{t=1}^{B} x_{\rho_i,t} \sum_{i'} \Pr[i' \prec i | t(i) = t] \left(\sum_{t'=t-v(i')}^{t} x_{\rho_{i'},t'} - \sum_{t'=t-u(i')}^{t} x_{\rho_{i'},t'}\right) \\
=&\sum_{t=1}^{B} x_{\rho_i,t} \sum_{i'} \Pr[i' \prec i | t(i) = t] \left(\sum_{t'=t-v(i')}^{t-u(i')-1} x_{\rho_{i'},t'}\right) \\
\geq& 0.
\end{aligned}$$

The second equality is due to $v(i') > u(i')$ for all i'. The last inequality is due the non-negative property of $x_{u,t}$. Hence, $\Pr[\omega(u)(i) = j] \geq \Pr[\omega(v)(i) = j]$ holds □

Proof of Theorem 7. The output r of Algorithm 4 satisfies $\mathbb{E}[f(r)] = \mathbb{E}[f(\omega(\sigma(x)))]$, and it is always feasible. By Lemma 8, ω is a monotone 1/2-CRS with respect to q, where q is the probability defined in Lemma 8. Moreover, $\sigma(x) \sim q$ holds. Hence, by Lemma 5, $\mathbb{E}[f(\omega(\sigma(x)))] \geq \mathbb{E}[f(\sigma(x))]/2$. Plugging in Lemma 6, we get $f_{\text{avg}}(\pi) = \mathbb{E}[f(r)] = \mathbb{E}[f(\omega(\sigma(x)))] \geq \bar{F}(\bar(x))/2$. □

Chapter 5: Conclusion

With the ever-increasing volume of data and the growing popularity of data centers, classic and well studied fields like job scheduling are revealing yet another fascinating perspective. In this thesis, we have studied various scheduling problems originating from the context of data center and cloud computing.

In Chapters 2 and 3, we studied a general framework that captures the application-level data communication patterns in data centers. Classical algorithms are not designed to take such data transfer into account. Chowdhury and Stoica [1] modeled this challenge as the coflow scheduling problem, which captures the scenario where data transfers occur in a uniform network. In Chapter 2, we follow the line of research from a theoretical perspective and improve on the approximation ratio. The improvement is achieved with a further exploitation of its relationship with the concurrent open shop problem, a relationship first discovered by Khuller and Purohit [14]. In addition to the LP based algorithm, we also developed a practically efficient algorithms based on the primal-dual technique, with the same theoretical guarantee. This theoretical result makes it way back to the system community [3], closing the loop of research from system to theory, and back to system. There is still a gap of two between upper and lower theoretical bounds, and we leave the closing

of this gap as an open question.

In Chapter 3, we consider the generalized case where data transfers happen on non-uniform or geo-distributed networks. In order to capture the sharing/merging/splitting capability of modern networks, we solve the problem via linear programming and multi-commodity flow, resulting in a nice heuristic that achieves close to optimality in experiments. The solution is further massaged to result in a tight approximation algorithm. Such a solution can be slow for large networks or a long period of time, but we believe that this result give the impetus for better approximation algorithms as well as fast heuristics that works well in practice.

While the previous chapters deal with the challenge originating from data transmission, in Chapter 4 we study the scheduling problem in the presence of cloud computing platforms. With rentable and revocable computing resources, another dimension is added to the classical scheduling problems: which instance to rent, and how to handle stochastic interruptions. We model and reduce this problem to the well studied correlated stochastic knapsack problem, replacing the target function with a submodular one, in order to capture the property of diminishing returns. An extra partition matriod constraint is added to capture budget cap and eliminate assumptions in previous works. For the reduced problem, we improved on the approximation ratio. There is still a gap comparing to the variant with linear target function, and we leave it as an open problem. This work is an attempt for the algorithmic challenges from rentable and revocable cloud servers, and we believe that it will inspire more work from both the theory and the system community.

5.1 Future Directions

In this section, we discuss the future directions of coflow scheduling and scheduling with spot instances.

5.1.1 Coflow Scheduling

Currently, the best approximation ratio for coflow scheduling in switch model is 5 approximation with release time and 4 approximation without release time [15]. On the hardness side, this problem generalizes concurrent open shop problem, which is NP-hard to get $(2 - \epsilon)$ approximation for any $\epsilon > 0$ [37, 38]. There is a gap of factor 2 in the approximation ratio. We want to either improve the approximation ratio or to get a tighter hardness bound. For the graph model, we currently have matching lower and upper bounds, but the solution uses time indexed LP which is not efficient to solve in practice. We hope to find some practically efficient algorithm which has the same or a slightly worse approximation bound.

While experiments show that our algorithm for the switch model works well in practice, there are several concerns. The first scenario that need attention is online setting: decisions need to be made before the next new job comes. In the work of Khuller et al. [29], a 12 adaptive online algorithm for coflow scheduling was given, and it can be improved to 6 adaptive if exponential time online algorithm is allowed. The next scenario is about special network structures. Real networks lie between switch model and graph model. Is it possible to get better approximation ratio in special graph types commonly found in datacenters, e.g., big tree, expander, etc.

The third scenario is a different target function. While weighted completion time is well studied target function, it is not fully satisfying in this situation. If a short job is released at time 1000, the optimal solution might schedule it right away, while an approximation algorithm targeting completion time would probably schedule at time 2000, getting a 2 approximation. The idea of flow time (also known as job lateness) is designed to address this: it denotes the difference between release time of a job and the time it finishes. Optimization for flow time proves hard [53, 54, 76, 77, 78, 79]. Recently, Batra et al. [53] made a breakthrough on flow time scheduling on single machine and got constant approximation ratio in pseudo-polynomial time. Feige et al. [54] improved the running time on this via black-box reduction and got a true polynomial time constant approximation algorithm. A nice direction would be follow up these work and extends to coflow scheduling, starting from concurrent open shop.

5.1.2 Scheduling Spot Instances

Comparing to previous works on correlated stochastic knapsack problems with linear target function, our work has a limitation from assumption we make. Despite the elimination of one assumption (see Section 4.2.4), the other assumption in Fukunaga et al. [28] remains. This assumption ensures a monotone property which is necessary for the contention resolution framework. Therefore, we would need a different technique to remove this constraint. Apart from this limitation, we lose another factor of 2 when rounding the solution, where in Ma [70], no factor is lost.

The reason behind the extra factor is partly due to the monotone property we just mentioned, and partly due to the complicated dependency in the rounding process of Ma [70]. An attempt to reduce or even remove this additional factor would help us both in the understanding of dependencies in the rounded solution, and better understanding of the contention resolution scheme.

In Ma [70], both the correlated stochastic knapsack problem and the MAB superprocess problem can be handled. However, our algorithm can only handle correlated stochastic knapsack problem. To be more precise, our proof technique will only work if for each arm in the MAB, for all but the transition after the first action, it happens with probability 0 or 1. In other words, there are no random events that are partially correlated with each other except for those related to the first action. It would be a great improvement if the fully generalized model in Ma [70] can be supported.

The last direction is to make the algorithm practical. Our algorithm is based on the multilinear-extension of submodular function. Such a function takes exponential time to evaluate, but can be approximated to arbitrary precision within polynomial time using sampling and concentration bounds. However, this is generally considered not very efficient in practice. To make things worse, the time indexed program and the budget cap add a factor of $\mathbb{N}^2$ to the size of the problem, leading to a more impractical algorithm. This can be partially fixed by limiting the set of possible budget caps. There is a line of research on monotone submodular optimization without multilinear-extension, which may be a promising direction to give algorithms that are efficient and applicable in practice.

Bibliography

[1] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[2] Hamidreza Jahanjou, Erez Kantor, and Rajmohan Rajaraman. Asymptotically optimal approximation algorithms for coflow scheduling. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 45–54. ACM, 2017.

[3] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia. In *Proceedings of the 2018 Conference of the Acm Special Interest Group on Data Communication*, 8 2018. doi: 10.1145/3230543.3230569. URL https://doi.org/10.1145/3230543.3230569.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. doi: 10.1145/1327452.1327492. URL https://doi.org/10.1145/1327452.1327492.

[5] Apache Software Foundation. Hadoop. URL https://hadoop.apache.org.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd Usenix Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[7] Shouxi Luo, Hongfang Yu, Yangming Zhao, Sheng Wang, Shui Yu, and Lemin Li. Towards practical and near-optimal coflow scheduling for data center networks. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3366–3380, 2016.

[8] Yupeng Li, Shaofeng H.-C. Jiang, Haisheng Tan, Chenzi Zhang, Guihai Chen, Jipeng Zhou, and Francis C. M. Lau. Efficient online coflow routing and scheduling. In *Proceedings of the 17th Acm International Symposium on Mobile Ad Hoc Networking and Computing - Mobihoc '16*, - 2016. doi: 10.1145/2942358.2942367. URL https://doi.org/10.1145/2942358.2942367.

[9] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. *SIGCOMM Comput. Commun. Rev.*, 44(4):443–454, August 2014.

ISSN 0146-4833. doi: 10.1145/2740070.2626315. URL https://doi.org/10.1145/2740070.2626315.

[10] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *SIGCOMM Comput. Commun. Rev.*, 45(4):393–406, August 2015. ISSN 0146-4833. doi: 10.1145/2829988.2787480. URL https://doi.org/10.1145/2829988.2787480.

[11] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *2015 Ieee Conference on Computer Communications (INFOCOM)*, 4 2015. doi: 10.1109/infocom.2015.7218408. URL https://doi.org/10.1109/infocom.2015.7218408.

[12] Ruozhou Yu, Guoliang Xue, Xiang Zhang, and Jian Tang. Non-preemptive coflow scheduling and routing. In *2016 Ieee Global Communications Conference (GLOBECOM)*, 12 2016. doi: 10.1109/glocom.2016.7842029. URL https://doi.org/10.1109/glocom.2016.7842029.

[13] Zhen Qiu, Cliff Stein, and Yuan Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 294–303, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3588-1. doi: 10.1145/2755573.2755592. URL http://doi.acm.org/10.1145/2755573.2755592.

[14] Samir Khuller and Manish Purohit. Brief announcement: Improved approximation algorithms for scheduling co-flows. In *Proceedings of the 28th Acm Symposium on Parallelism in Algorithms and Architectures*, pages 239–240, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4210-0. doi: 10.1145/2935764.2935809. URL https://doi.org/10.1145/2935764.2935809.

[15] Saba Ahmadi, Samir Khuller, Manish Purohit, and Sheng Yang. On scheduling coflows. *Algorithmica*, 82(12):3604–3629, 2020. doi: 10.1007/s00453-020-00741-3. URL https://doi.org/10.1007/s00453-020-00741-3.

[16] Mehrnoosh Shafiee and Javad Ghaderi. An improved bound for minimizing the total weighted completion time of coflows in datacenters. *IEEE/ACM Transactions on Networking*, 26(4):1674–1687, 2018. doi: 10.1109/tnet.2018.2845852. URL https://doi.org/10.1109/tnet.2018.2845852.

[17] Jie You and Mosharaf Chowdhury. Terra: Scalable cross-layer gda optimizations. https://arxiv.org/abs/1904.08480, 2019.

[18] Amazon. Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot/.

[19] Microsoft. Microsoft low priority vm. https://azure.microsoft.com/en-us/pricing/details/batch/.

[20] Google. Preemptible vm instances. `https://cloud.google.com/compute/docs/instances/preemptible`.

[21] Adish Singla, Sebastian Tschiatschek, and Andreas Krause. Noisy submodular maximization via adaptive sampling with applications to crowdsourced image collection summarization. In *Proceedings of the Thirtieth Aaai Conference on Artificial Intelligence*, AAAI'16, pages 2037–2043. AAAI Press, 2016.

[22] Andreas Krause and Carlos Guestrin. Submodularity and its applications in optimized information gathering. *ACM Transactions on Intelligent Systems and Technology*, 2(4):1–20, 2011. doi: 10.1145/1989734.1989736. URL `https://doi.org/10.1145/1989734.1989736`.

[23] Rishabh Iyer and Jeff Bilmes. Submodular optimization with submodular cover and submodular knapsack constraints. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2436–2444, Red Hook, NY, USA, 2013. Curran Associates Inc.

[24] Alan Kuhnle, J. David Smith, Victoria Crawford, and My Thai. Fast maximization of non-submodular, monotonic functions on the integer lattice. volume 80 of *Proceedings of Machine Learning Research*, pages 2786–2795, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/kuhnle18a.html`.

[25] Tasuku Soma and Yuichi Yoshida. Non-monotone dr-submodular function maximization. In *Proceedings of the Thirty-First Aaai Conference on Artificial Intelligence*, AAAI'17, pages 898–904. AAAI Press, 2017.

[26] Ryan Gomes and Andreas Krause. Budgeted nonparametric learning from data streams. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 391–398, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

[27] Sheng Yang, Sunav Chodhary, Subrata Mitra, Kanak Mahadik, and Samir Khuller. Distributed ml on aws spot instances. unpublished manuscript, 2019.

[28] Takuro Fukunaga, Takuya Konishi, Sumio Fujita, and Ken ichi Kawarabayashi. Stochastic submodular maximization with performance-dependent item costs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:1485–1494, 2019. doi: 10.1609/aaai.v33i01.33011485. URL `https://doi.org/10.1609/aaai.v33i01.33011485`.

[29] Samir Khuller, Jingling Li, Pascal Sturmfels, Kevin Sun, and Prayaag Venkat. Select and permute: An improved online framework for scheduling to minimize weighted completion time. *Theoretical Computer Science*, 795:420–431, 2019.

[30] Ruijiu Mao, Vaneet Aggarwal, and Mung Chiang. Stochastic non-preemptive co-flow scheduling with time-indexed relaxation. *IEEE International Conference on Computer Communications*, 2018.

[31] Zhi-Long Chen and Nicholas G Hall. Supply chain scheduling: Conflict and cooperation in assembly systems. *Operations Research*, 55(6):1072–1089, 2007.

[32] Naveen Garg, Amit Kumar, and Vinayaka Pandit. *Order Scheduling Models: Hardness and Algorithms*, pages 96–107. FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-77050-3_8. URL https://doi.org/10.1007/978-3-540-77050-3_8.

[33] Joseph Y-T Leung, Haibing Li, and Michael Pinedo. Scheduling orders for multiple product types to minimize total weighted completion time. *Discrete Applied Mathematics*, 155(8):945–970, 2007.

[34] Monaldo Mastrolilli, Maurice Queyranne, Andreas S Schulz, Ola Svensson, and Nelson A Uhan. Minimizing the sum of weighted completion times in a concurrent open shop. *Operations Research Letters*, 38(5):390–395, 2010.

[35] Guoqing Wang and TC Edwin Cheng. Customer order scheduling to minimize total weighted completion time. *Omega*, 35(5):623–626, 2007.

[36] Sungjin Im, Benjamin Moseley, Kirk Pruhs, and Manish Purohit. Matroid coflow scheduling. *International Colloquium on Automata, Languages and Programming*, 2019.

[37] Nikhil Bansal and Subhash Khot. Inapproximability of hypergraph vertex cover and applications to scheduling problems. In *International Colloquium on Automata, Languages and Programming*, pages 250–261. Springer, 2010.

[38] Sushant Sachdeva and Rishi Saket. Optimal inapproximability for scheduling problems via structural hardness for hypergraph vertex cover. In *IEEE Conference on Computational Complexity*, pages 219–229. IEEE, 2013.

[39] Maurice Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*, 58(1-3):263–285, 1993.

[40] James M Davis, Rajiv Gandhi, and Vijay H Kothari. Combinatorial algorithms for minimizing the weighted sum of completion times on a single machine. *Operations Research Letters*, 41(2):121–125, 2013.

[41] Julia Chuzhoy, Venkatesan Guruswami, Sanjeev Khanna, and Kunal Talwar. Hardness of routing with congestion in directed graphs. In *ACM Symposium on Theory of Computing*, pages 165–178. ACM, 2007.

[42] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, pages 15–26. ACM, 2013.

[43] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, pages 3–14. ACM, 2013.

[44] Intel Hadoop. Big data benchmark for big bench. `https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench`, 2016.

[45] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 1049–1058. VLDB Endowment, 2006.

[46] Meikel Poess and Chris Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[47] Facebook. Statistical workload injector for mapreduce (swim). `https://github.com/SWIMProjectUCB/SWIM`, 2014.

[48] Mosharaf Chowdhury. Coflow benchmark based on facebook traces. `https://github.com/coflow/coflow-benchmark`, 2015.

[49] Sungjin Im, Maxim Sviridenko, and Ruben Van Der Zwaan. Preemptive and non-preemptive generalized min sum set cover. *Mathematical Programming*, 145(1-2):377–401, 2014.

[50] Maurice Queyranne and Maxim Sviridenko. A (2+ ε)-approximation algorithm for the generalized preemptive open shop problem with minsum objective. *Journal of Algorithms*, 45(2):202–212, 2002.

[51] Andreas S. Schulz and Martin Skutella. *Random-based scheduling new approximations and LP lower bounds*, pages 119–133. Randomization and Approximation Techniques in Computer Science. Springer Berlin Heidelberg, 1997. doi: 10.1007/3-540-63248-4_11. URL `https://doi.org/10.1007/3-540-63248-4_11`.

[52] Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2018. URL `http://www.gurobi.com`.

[53] Jatin Batra, Naveen Garg, and Amit Kumar. Constant factor approximation algorithm for weighted flow time on a single machine in pseudo-polynomial time. In *Foundations of Computer Science*, pages 778–789. IEEE, 2018.

[54] Uriel Feige, Janardhan Kulkarni, and Shi Li. *A Polynomial Time Constant Approximation For Minimizing Total Weighted Flow-time*, pages 1585–1595. Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2019. doi: 10.1137/1.9781611975482.96. URL `https://doi.org/10.1137/1.9781611975482.96`.

[55] Maria Carla Calzarossa, Marco L. Della Vedova, Luisa Massari, Dana Petcu, Momin I. M. Tabash, and Daniele Tessera. *Workloads in the Clouds*, pages 525–550. Springer International Publishing, Cham, 2016. doi: 10.1007/978-3-319-30599-8_20.

[56] Prasad Saripalli, G. V. R. Kiran, R. Ravi Shankar, Harish Narware, and Nitin Bindal. Load prediction and hot spot detection models for autonomic cloud computing. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 397–402, Washington, DC, USA, 2011. IEEE Computer Society. doi: 10.1109/UCC.2011.66.

[57] Pradeep Ambati and David Irwin. Optimizing the cost of executing mixed interactive and batch workloads on transient vms. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '19, pages 45–46, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6678-6. doi: 10.1145/3309697.3331489.

[58] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 121–134, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6011-1. doi: 10.1145/3267809.3267819.

[59] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 575–588, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4938-3. doi: 10.1145/3064176.3064181.

[60] Prateek Sharma, David Irwin, and Prashant Shenoy. Portfolio-driven resource management for transient cloud servers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):1–23, 2017. doi: 10.1145/3084442. URL `https://doi.org/10.1145/3084442`.

[61] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 484–496, New York, NY, USA, 2016. ACM. doi: 10.1145/2987550.2987576.

[62] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 589–604. ACM, 2017.

[63] Amazon Inc. Spot Instance Advisor. `https://aws.amazon.com/ec2/spot/instance-advisor/`, 2019.

[64] Sébastien Bubeck et al. Convex Optimization: Algorithms and Complexity. *Foundations and Trends in Machine Learning*, 8(3-4):231–357, 2015. ISSN 1935-8237. doi: 10.1561/2200000050.

[65] Prateek Jain and Purushottam Kar. Non-convex Optimization for Machine Learning. *Foundations and Trends in Machine Learning*, 10(3-4):142–363, 2017. doi: 10.1561/2200000058.

[66] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618, 9780262035613.

[67] Xiaoxi Zhang, Jianyu Wang, Fanjing Wu, Gauri Joshi, and Carlee Joe-Wong. Machine learning on the cheap: An optimized strategy to exploit spot instance. `https://www.andrew.cmu.edu/user/gaurij/spot_instance_ml.pdf`, 2019.

[68] Anupam Gupta, Ravishankar Krishnaswamy, Marco Molinaro, and R Ravi. Approximation algorithms for correlated knapsacks and non-martingale bandits. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 827–836. IEEE, 2011.

[69] Will Ma. Improvements and generalizations of stochastic knapsack and multi-armed bandit approximation algorithms. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1154–1163. Society for Industrial and Applied Mathematics, 2014.

[70] Will Ma. Improvements and generalizations of stochastic knapsack and markovian bandits approximation algorithms. *Mathematics of Operations Research*, 43(3):789–812, 2018. doi: 10.1287/moor.2017.0884. URL `https://doi.org/10.1287/moor.2017.0884`.

[71] Arash Asadpour and Hamid Nazerzadeh. Maximizing stochastic monotone submodular functions. *Management Science*, 62(8):2374–2391, 2016. doi: 10.1287/mnsc.2015.2254. URL `https://doi.org/10.1287/mnsc.2015.2254`.

[72] Moran Feldman, Joseph Naor, and Roy Schwartz. A unified continuous greedy algorithm for submodular maximization. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, 10 2011. doi: 10.1109/focs.2011.46. URL `https://doi.org/10.1109/focs.2011.46`.

[73] Gruia Calinescu, Chandra Chekuri, Martin Pál, and Jan Vondrák. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM Journal on Computing*, 40(6):1740–1766, 2011. doi: 10.1137/080733991. URL `https://doi.org/10.1137/080733991`.

[74] Chandra Chekuri, Jan Vondrák, and Rico Zenklusen. Submodular function maximization via the multilinear relaxation and contention resolution schemes. *SIAM Journal on Computing*, 43(6):1831–1879, 2014. doi: 10.1137/110839655. URL `https://doi.org/10.1137/110839655`.

[75] M. Feldman. *Maximization Problems with Submodular Objective Functions*. Ph.d. dissertation,, Technion - Israel Institute of Technology, 2013.

[76] Hans Kellerer, Thomas Tautenhahn, and Gerhard Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM Journal on Computing*, 28(4):1155–1166, 1999.

[77] Nikhil Bansal and Kirk Pruhs. The geometry of scheduling. *SIAM Journal on Computing*, 43(5):1684–1698, 2014. doi: 10.1137/130911317. URL `https://doi.org/10.1137/130911317`.

[78] Nikhil Bansal and Ho-Leung Chan. Weighted flow time does not admit o(1)-competitive algorithms. In *Proceedings of the Twentieth Annual Acm-Siam Symposium on Discrete Algorithms*, 1 2009. doi: 10.1137/1.9781611973068.134. URL `https://doi.org/10.1137/1.9781611973068.134`.

[79] Nikhil Bansal and Kedar Dhamdhere. Minimizing weighted flow time. *ACM Transactions on Algorithms*, 3(4):39, 2007. doi: 10.1145/1290672.1290676. URL `https://doi.org/10.1145/1290672.1290676`.