Resource Allocation In Large-Scale Distributed Systems

Mehrnoosh Shafieezade Abade

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

Resource allocation in large-scale distributed systems

Mehrnoosh Shafieezade Abade

The focus of this dissertation is design and analysis of scheduling algorithms for distributed computer systems, i.e., data centers. Today's data centers can contain thousands of servers and typically use a multi-tier switch network to provide connectivity among the servers. Data centers are the host for execution of various data-parallel applications. As an abstraction, a job in a data center can be thought of as a group of interdependent tasks, each with various requirements which need to be scheduled for execution on the servers and the data flows between the tasks that need to be scheduled in the switch network. In this thesis, we study both flow and task scheduling problems under the features of modern parallel computing frameworks.

For the flow scheduling problem, we study three models. The first model considers a general network topology where flows among the various source-destination pairs of servers are generated dynamically over time. The goal is to assign the end-to-end data flows among the available paths in order to efficiently balance the load in the network. We propose a myopic algorithm that is computationally efficient and prove that it asymptotically minimizes the total network cost using a convex optimization model, fluid limit and Lyapunov analysis. We further propose randomized versions of our myopic algorithm. The second model consider the case that there is dependence among flows. Specifically, a *coflow* is defined as a collection of parallel flows whose completion time is determined by the completion time of the last flow in the collection. Our main result is a 5-approximation deterministic algorithm that schedule coflows in polynomial time so as to minimize the total weighted completion times. The key ingredient of our approach is an improved linear program formulation for sorting the coflows followed by a simple list scheduling policy. Lastly, we study scheduling coflows of multi-stage jobs to minimize the jobs' total weighted completion times. Each job is represented by a DAG (*Directed Acyclic Graph*) among its coflows that captures the dependencies among the coflows. We define $g(m) = \log(m)/\log(\log(m))$ and $h(m, \mu) = \log(m\mu)/(\log(\log(m\mu))$, where m is number of servers, μ is the maximum number of coflows in a job. We develop two algorithms with approximation ratios $O(\sqrt{\mu}g(m))$ and $O(\sqrt{\mu}g(m)h(m, \mu))$ for jobs with general DAGs and rooted trees, respectively. The algorithms rely on random delaying and merging optimal schedules of the coflows in the jobs' DAG, followed by enforcing dependency among coflows and the links' capacity constraints.

For the task scheduling problem, we study two models. We consider a setting where each job consists of a set of parallel tasks that need to be processed on different servers, and the job is completed once all its tasks finish processing. In the first model, each job is associated with a utility which is a decreasing function of its completion time. The objective is to schedule tasks in a way that achieves max-min fairness for jobs' utilities.We first show a strong result regarding NP-hardness of this problem. We then proceed to define two notions of approximation solutions and develop scheduling algorithms that provide guarantees under these approximation notions, using dynamic programming and random perturbation of tasks' processing times. In the second model, we further assume that processing times of tasks can be server dependent and a server can process (pack) multiple tasks at the same time subject to its capacity. We then propose three algorithms with approximation ratios of 4, $(6 + \epsilon)$, and 24 for different cases where preemption and migration of tasks among the servers are or are not allowed. Our algorithms use a combination of linear program relaxation and greedy packing techniques.

To demonstrate the gains in practice, we evaluate all the proposed algorithms and compare their performances with the prior approaches through extensive simulations using real and synthesized traffic traces. We hope this work inspires improvements to existing job management and scheduling in distributed computer systems.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Columbia University and the Electrical Engineering Department for providing me the opportunity to pursue my PhD. Particularly, I would like to express my sincere gratitude to my advisor Prof. Javad Ghaderi for his supervision and support in the course of my PhD studies. He created a great research environment through his scientific enthusiasm and discipline. He patiently taught me how to think critically, communicate my ideas, and anything I needed to become an independent researcher.

I would like to thank the members of my defense committee, Professor Clifford Stein, Professor Vishal Misra, Professor Ethan Katz-Bassett, and Dr. Vahab Mirrokni for their time and thoughtful comments. I am also thankful to Mrs. Elsa Sanchez and Mr. Dennis Scott-Torbet for always being my first points of contact in the Electrical Engineering Department whenever I had a question or needed any help.

I feel very lucky to have great friends who care about me and make my life more fun and enjoyable. My special gratitude goes to Jeanne and Kevin for being so supportive and kind to me. They have been there for me in my hardest times in New York City far away from my family. I would also like to thank Zahra, Parima, Niloofar, Niloofar, AmirHossein, Christos, and many more with whom I share a lot of great memories. Further, I am immensely thankful to Golnaz, my generous and knowledgable friend and physical therapist who gave me hope and saved my life. I cannot thank her enough.

Foremost, I express my deepest sense of gratitude to my parents, GholamAli and Monireh, for their unconditional love, care, and dedication. Any accomplishment in my life was not possible without their willingness to always sacrifice their comfort for me and my brother. I am specially thankful to them for raising me confident and independent, and for their constant encouragement. I would also like to thank Soroosh, my kind, thoughtful, and positive brother. I am so lucky to have someone who is always there to listen to me and support me. This work is dedicated to my parents and my brother.

Dedication

To Maman, Baba, and Soroosh...

Chapter 1: Introduction

Data centers have received significant attention as a cost-effective infrastructure for storing large volumes of data and supporting large-scale Internet services by Google, Amazon, Facebook, etc. Data centers also serve numerous number of small and medium sized organizations for their requirements such as financial operations, data analysis, and scientific computations. The increasing popularity of cloud-computing services such as Microsoft Azure, Amazon Web Services, and many others has also contributed to the increasing growth of data centers. A key challenge in the data centers is to efficiently support a wide range of "jobs" (i.e, queries, log analysis, machine learning, graph processing, stream processing, etc.) on their physical platform. These jobs routinely process Peta bytes of data on thousands of machines (servers) every day.

As an abstraction, a job can be thought of as a group of interdependent tasks, each with various requirements of CPU, memory, disk, network bandwidth, etc., which need to be scheduled for execution on the servers and the data flows between the tasks need to be scheduled in the switch network. One can view this complex scheduling problem as a joint scheduling of tasks in servers and scheduling of flows (rate assignment known as congestion control and flow routing inside the data center network), as depicted in Figure 1.1. Solving this scheduling problem efficiently is very challenging due to the coupling across server and network resources, and co-existence of variety of applications with very diverse requirements, from low latency to high throughput, all running in the same data center cluster. A practical scheduler also needs to be scalable in terms of the size of the data center [1, 2, 3].

As a first approach toward dealing with this complicated problem, we can consider task scheduling and flow scheduling problems separately. Although this approach can be suboptimal, it helps with understanding the structure of possible solutions which can be subsequently used in solving the joint problem. Each of these separated problems still remains very complex, and in fact

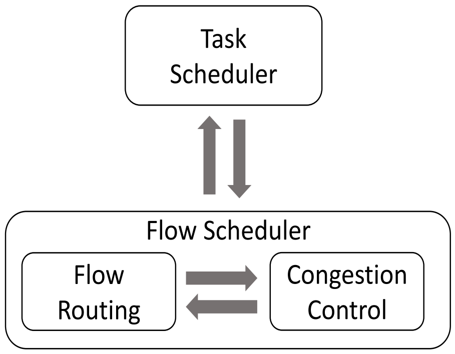


Figure 1.1: Scheduling problems in data centers.

even simplified off-line versions of many of them, when all the information is available, are hard combinatorial problems (e.g. NP-hard).

1.1 Flow Scheduling

Scheduling flows in the data center multi-tier switch network is a challenging and significantly important problem. This is because while the traffic generated by the servers is growing exponentially over time (it is doubling every 12-15 months [3]), provided bandwidth by the data center topology is not growing that fast and remains the main bottleneck [2]. Such communications can account for a large portion of the job completion time, and hence can have a significant impact on application latency. Moreover, various traffic patterns such as one-to-one, one-to-many, and many-to-many can arise as a result of running data-parallel computing jobs [4, 5]. Further, data center traffic is very bursty and unpredictable which makes the problem even harder. To address this problem, there has been significant research on designing new networks with better topological features that are scalable and cost efficient. In addition, there has been a parallel line of research on flow scheduling algorithms that take into account the application requirements and make better use of network resources.

1.1.1 Load Balancing in A General Network Topology

Chapter 2 is dedicated to our work on the design and analysis of algorithms for scheduling flows among servers. We consider a general network topology where each link has a cost which is a convex function of the link congestions (e.g. this could be a link latency measure). We propose a low complexity, congestion-aware algorithm that routes the flows in an online fashion and without splitting. Our algorithm assigns every arriving flow to an available path with the minimum marginal cost (congestion). We further prove that it asymptotically minimizes the total network cost. Further, we use extensive simulations to test the performance of our algorithms under a wide range of traffic conditions and different data center architectures. This algorithm needs to consider all the available paths for an arriving flow and finds the shortest path based on the marginal cost of paths. To address this issue, we describe and empirically evaluate randomized versions of our algorithm which have less complexity than the original algorithm, while can effectively provide a large fraction of the performance gain obtained by the original algorithm. Our approach is motivated by the literature on randomized load balancing for scheduling jobs in servers [6].

1.1.2 Coflow Scheduling

In the next chapter, Chapter 3, we consider scheduling flows in data center networks in presence of dependence among flows. Many data-parallel applications (e.g. MapReduce [5], Dryad [6], etc.) consist of multiple computation and communication stages or have machines grouped by functionality. While computation involves local operations in servers, communication takes place at the level of machine groups and involves transfer of many pieces of intermediate data across groups of machines for further processing. A computation stage often cannot start or be completed unless all the required data pieces from the preceding communication stage are received. Therefore, the application latency is determined by the transfer of the last flow between the groups. Hence, to meet application level requirements, this is crucial to take into account the dependence among flows [7]. *Coflow* is an abstraction to capture these communication patters [8]. Specifically, a coflow is defined as a collection of parallel flows whose completion time is determined by the completion time of the last flow in the collection. We study the algorithmic task of determining when to start serving each flow and at what rate, in order to minimize the weighted average completion time of coflows in the system. Our main result is a polynomial-time deterministic algorithm based on a linear program followed by a simple list scheduling policy with approximation ratio of 5, which improves the prior best known ratio of 12. Further, this is currently the best approximation algorithm for this problem. Extensive simulation results are also presented that verify the performance of our algorithm and show improvements over the prior approaches.

1.1.3 Scheduling Coflows of Multi-Stage Jobs

Finally, motivated by the fact that applications in data-parallel computing typically consist of multiple stages, we consider scheduling coflows of multi-stage jobs in Chapter 4. Each multi-stage job is represented by a DAG (*Directed Acyclic Graph*) among its coflows that capture their dependencies. For jobs with a single communication stage, minimizing the average completion times of coflows results in the job's latency improvement. However, for multi-stage jobs, minimizing the average coflow completion time might not be the right metric and might even lead to a worse performance, as it ignores the dependencies between coflows in a job [9, 10]. Our goal is to minimize the total weighted completion time of jobs, where the completion time of a job is determined by the completion of the last coflow in its DAG. We prove that even for a single multi-stage job represented by a rooted tree, it is NP-hard to schedule it so as to minimize its completion time. We then develop an $O(\mu g(m))$ approximation algorithm for minimizing the total weighted completion time of a given set of multi-stage jobs, where m is number of servers in the system, μ is the maximum number of coflows in a job, and $g(m) = \log(m)/\log(\log(m))$. When the jobs' DAGs are rooted trees, we improve this result and get an $O(\sqrt{\mu}g(m)h(m, \mu))$ approximation algorithm, where $h(m, \mu) = \log(m\mu)/(\log(\log(m\mu))$. Our algorithms rely on random delaying and merging optimal schedules of the coflows in the job's DAG, followed by enforcing dependency among coflows and the links' capacity constraints. These results exponentially improve the previous $O(m)$ algorithm proposed in [9]. We further show the improvement of our algorithms' performance in comparison with previous algorithms through extensive simulations.

1.2 Task Scheduling

Next, we focus on the problem of scheduling tasks with heterogeneous resource requirements in a cluster of servers. Our models are motivated by modern parallel computing frameworks, e.g. Hadoop and Spark [11, 12] that have enabled large-scale data processing in computing clusters. In such frameworks, the data is typically distributed across a cluster of servers and is processed in multiple stages. In each stage, a set of tasks are executed on the machines where each task is preferred to be scheduled on one of the machines that has its required data block [4, 13] (a.k.a. *data locality*). The tasks in a stage can run in parallel, however, the job is finished or the next stage can start once all the tasks in the preceding stage(s) are completed [4, 5, 14]. We refer to such constraints as *synchronization* constraints. Another main feature of parallel computing clusters is that jobs can have diverse tasks and processing requirements. This has been further amplified by the increasing complexity of workloads, i.e., from traditional batch jobs, to queries, graph processing, streaming, and machine learning jobs, that all need to share the same cluster. Despite the vast scheduling literature, scheduling algorithms with theoretical results (approximation ratios) are mainly based on simple models that assume each job is only one task (ignoring dependency among tasks and their collective impact on the job's completion time), or tasks are processed on any server arbitrarily (ignoring data locality). Our goal is to design scheduling algorithms, with theoretical guarantees, under the features of modern parallel computing clusters.

1.2.1 Max-Min Fairness of Completion Times

We first study the max-min fairness of multi-task jobs in distributed computing platforms in Chapter 5. We consider a setting where each job consists of a set of parallel tasks that need to be processed on different servers, and the job is completed once all its tasks finish processing. Each job is associated with a utility which is a decreasing function of its completion time, and captures how sensitive it is to latency. The objective is to schedule tasks in a way that achieves max-min fairness for jobs' utilities, i.e., an optimal schedule in which any attempt to improve the utility of a job necessarily results in hurting the utility of some other job with smaller or equal utility. We first show a strong result regarding NP-hardness of finding the max-min fair vector of job utilities. The implication of this result is that achieving max-min fairness in many other distributed scheduling problems (e.g., coflow scheduling) is NP-hard. We then proceed to define two notions of approximation solutions: one based on finding a certain number of elements of the max-min fair vector, and the other based on a single-objective optimization whose solution gives the max-min fair vector. We develop scheduling algorithms that provide guarantees under these approximation notions, using dynamic programming and random perturbation of tasks' processing times. We verify the performance of our algorithms through extensive simulations, using a real traffic trace from a large Google cluster.

1.2.2 Minimizing Weighted Average of Completion Times

We consider the objective of minimizing the weighted average of jobs' completion times in Chapter 6. Besides the *synchronization* constraint, we consider a generalized version of the *data locality* constraint in this model. More precisely, we assume that assignment of tasks to servers is subject to *placement* constraints, i.e., each task can be processed only on a subset of servers, and processing times can also be server dependent. We further take into the account the fact that a server can process (*pack*) multiple tasks at the same time, however the cumulative resource requirement of the tasks should not exceed the server's capacity. We consider both preemptive and non-preemptive scheduling. In a *non-preemptive* schedule, a task cannot be preempted (and hence cannot be migrated among servers) once it starts processing on a server until it is completed. In a *preemptive* schedule, a task may be preempted and resumed later in the schedule, and we further consider two cases depending on whether migration of a task among servers is allowed or not. For the case that migration of tasks among the placement-feasible machines is allowed, we propose a preemptive algorithm with an approximation ratio of $(6 + \epsilon)$. In the special case that only one machine can process each task, we design an algorithm with improved approximation ratio of 4.

Finally, in the case that migrations (and preemptions) are not allowed, we design an algorithm with an approximation ratio of 24. Our algorithms use a combination of linear program relaxation and greedy packing techniques. We present extensive simulation results, using a real traffic trace from a large Google cluster, that demonstrate that our algorithms yield significant gains over the prior approaches.

Chapter 2: Load Balancing in A General Network Topology

2.1 Introduction

There has been a dramatic shift over the recent decades with search, storage, and computing moving into large-scale data centers. Today's data centers can contain thousands of servers and typically use a multi-tier switch network to provide connectivity among the servers. To maintain efficiency and quality of service, it is essential that the data flows among the servers are mapped to the available paths in the network properly in order to balance the load and minimize the cost (e.g., delay, congestion, etc.). For example when a large flow is routed poorly, collision with the other flows can cause some links to become congested, while other less utilized paths are available.

The data center networks rely on path multiplicity to provide scalability, flexibility, and cost efficiency. Consequently, there has been much research on flow scheduling algorithms that make better use of the path multiplicity (e.g., [2, 15, 16, 17, 18]) or designing new networks with better topological features (e.g., FatTree [2], VL2 [19], hypercube [20], hypergrid [21], random graphs such as JellyFish [22], etc.).

In this chapter, we consider a general network topology where each link is associated with a cost which is a convex function of the link utilization (e.g., this could be a latency function). The network cost is defined as the sum of the link costs. Flows among the various source-destination pairs are generated dynamically over time where each flow is associated with a size (rate) and a duration. Once a flow is assigned to a path in the network, it consumes resource (bandwidth) equal to its size (rate) from all the links along its path for its duration. The main question that we ask is the following. Is it possible to design a low-complexity algorithm, that assigns the flows to the available paths in an *online fashion* and *without splitting*, so as to minimize the average network cost?

In general, multi flow routing in networks has been extensively studied from both networking systems and theoretical perspective, however multi flow routing considered in this chapter has two key distinguishing objectives:

1. *it does not allow flow splitting* because splitting the flow is undesirable due to TCP reordering effect [23]. Resolving packet reordering requires modification of protocol stack [24], which might be costly. Without splitting, many versions of multi flow routing in networks become hard combinatorial problems [25, 26]. In fact, the static version of the problem considered in this chapter (i.e., given a static list of flows, assigning flows to paths without splitting so as to balance the load in the network) is known to be NP-hard, through its connection to the Partition problem [27][1].

2. *it allows dynamic routing* because it considers the current utilization of links in the network when making the routing decisions for newly arrived flows unlike static solutions where the mapping of flows to the paths is fixed and requires the knowledge of the traffic matrix.

2.1.1 Related Work

Seminal solutions for flow routing in data centers (e.g. [19, 28]) rely on Equal Cost Multi Path (ECMP) load balancing which statically splits the traffic among available shortest paths (via flow hashing). However, it is well known [16, 15, 18, 17, 29] that ECMP can balance load poorly since it may map large long-lived flows to the same path, thus causing significant load imbalance. Further, ECMP is suited for symmetric architectures such as FatTree and performs poorly in presence of asymmetry either due to link failures [30] or in recently proposed data center architectures [22]. Theoretical performance of ECMP in Clos networks under a static flow model has been studied in [31]. There have been recent efforts to address the shortcomings of ECMP. The proposed algorithms range from centralized solutions (e.g., [15, 16]), where a centralized scheduler makes routing decisions based on global view of the network, to distributed solutions (e.g., [18, 32])

[1]In the Partition problem, given a set of numbers, we are asked to divide them into two subsets such that the maximum of the sum of the numbers in the sets is minimized. This can be reformulated as the load balancing in a simple two-node network with two parallel edges.

where routing decisions are made in a distributed manner by the switches. There are also host-based protocols based on Multi Path TCP (e.g., [17]) where the routing decisions are made by the end-host transport protocol rather than by the network operator; however, they require significant changes to Transport layer which might not be feasible in public cloud platforms [24]. Authors in [33] investigated a more general problem based on a Gibbs sampling technique and proposed a plausible heuristic that requires re-routing and interruption of flows (which is operationally expensive). There are also algorithms that allow flow splitting and try to resolve the packet reordering effect in *symmetric* network topologies [32, 24, 34]. As explained, dealing with packet reordering involves overhead and modification of protocol stack.

Our work is also related to a large body of literature on traffic engineering and congestion control. For brevity, we only highlight the most relevant work. The first line of work, e.g. [35, 36, 37], studies the problem of minimizing the cost of carrying traffic in a static multi-commodity flow model and under a convex cost function for the link rates. *Given the knowledge of the traffic matrix* (commodities) among the nodes, routing algorithms are proposed that iteratively update *the fraction of traffic of each flow* that should be sent on each outgoing link in the network. They rely on splitting flows among the least weighted paths where the weight of each link is defined by its marginal link cost.

The second line of work is atomic and non-atomic congestion games in game theory [38, 39, 40, 41]. In the context of routing, players are the commodities, strategy sets are the set of directed source-destination paths for the commodities, the edge cost $c_e(f_e)$ is a function of the amount of congestion f_e over edge e, and the path cost $c_p(f)$ is the sum of the cost of the links along the path p. A player i incurs a cost $c_p(f)f_p^{(i)}$ for sending $f_p^{(i)}$ amount of traffic over the path p. In the atomic games, each player must choose a single path to route its commodity, while in non-atomic games, player can distribute its commodity fractionally over the set of paths. The two versions are fundamentally different. While the atomic game in general does not admit a Nash equilibrium, the nonatomic game always has a Nash equilibrium (Wardrop equilibrium) [42]. In Wardrop equilibrium, all the paths used by a given commodity have equal cost. Moreover, it's known in non-atomic games that selfish best response moves (selfish routing) by the players iteratively converge to the Wardrop equilibrium, which is a local minimum of a potential function (network cost) $\sum_e \int_0^{f_e} c_e(x)dx$.

The third line of work is oblivious routing [43, 44, 45] in which routes are computed to optimize the worst-case performance over the set of traffic matrices. This ensures that the computed routes are prepared for changes in traffic demands without the need to update the routes, however this is a pessimistic point of view and may be far from optimal in relatively stable periods of traffic or stable networks [44].

While the proposed myopic algorithm in this chapter is reminiscent of prior algorithms under flow splitting and non-atomic games (e.g. [35, 36, 37, 42, 40, 41]), the results in this chapter are not trivially drawn from these prior work. First, unlike [35, 36, 37, 42, 40, 41] that rely on splitting flows in any granularity and rerouting them continuously to find the optimal routing, we *do not allow flow splitting and migrations*. Second, unlike [35, 36, 37, 42, 40, 41] that consider a static set of flows with known traffic demand, we are dealing with a *dynamic* version of the problem when flows arrive and depart dynamically over time and the traffic demand *is not* known. Such constraints arise in practice due to the varying nature of the traffic over time and space in data centers as well as undesirability of packet reordering in flow splitting. Our technical approach relies on a careful analysis of the fluid limits of the system under the myopic policy (without flow splitting) and proof of convergence to an invariant set which is the set of optimal flow assignments in steady state. Under unsplittable flows, the fluid limits are not continuously differentiable which poses a significant technical challenge. Intuitively, as the number of flows in the system grows, the difference between the optimal expected network cost under unsplittable flow assignment and that under splittable flow assignment should vanish in the performance ratio. We rigorously establish this intuition, and further, present deterministic and randomized algorithms with low complexity which perform very well in practice.

Finally, Software Defined Networking (SDN) has enabled network control with quicker and more flexible adaptation to changes in the network topology or the traffic pattern and can be leveraged to implement centralized or hybrid algorithms in data centers [2, 46, 47, 48]. The weight construct in the algorithms proposed in this chapter can provide an approach to optimally accommodate dynamic variations in data center network traffic in centralized control platforms such as OpenFlow [46].

2.1.2 Contributions

The main contributions of this chapter can be summarized as follows.

- **Asymptotic optimality of a myopic algorithm.** We propose and analyze a simple flow scheduling algorithm to minimize the average network cost (the sum of convex functions of link utilizations). Specifically, we propose a myopic algorithm that assigns every arriving flow to an available path with the minimum marginal cost (i.e., the path which yields the minimum increase in the network cost after assignment). We prove that this simple myopic algorithm is asymptotically optimal in *any* network topology, in the sense that *the performance ratio between the average network cost under the myopic algorithm and the optimal cost approaches* 1 *as the mean number of flows in the system increases*. The myopic algorithm does not rely on flow splitting, hence packets of the same flow will travel along the same path without reordering. Further, it does not require migration/rerouting of the flows or the knowledge of the traffic pattern.

- **A low complexity randomized algorithm.** We also propose randomized versions of our myopic algorithm which have much lower complexity. In the randomized algorithm with parameter $k \geq 2$, instead of considering all the available paths upon arriving of a flow, k paths are chosen at random and then the flow is assigned to the path with the minimum marginal cost among these k paths. Similar to the myopic algorithm, randomized versions do not rely on flow splitting, flow migration/rerouting, or the knowledge of the traffic pattern. We empirically investigate the effect of parameter k on the algorithm performance.

- **Empirical evaluation of the algorithms.** We evaluate our myopic algorithm and its randomized versions under various workload and network topologies. For the flow generation, we consider two traffic models: (i) Poisson arrival of flows with exponentially distributed durations, and (ii) based on data from empirical studies of data center traffic. For the network topology, we consider FatTree (a highly structured topology), and JellyFish (a random topology). Our empirical results show that the myopic algorithm in fact performs very well under a wide range of traffic conditions in both data center topologies. Further, the randomized algorithms can perform very well by choosing the proper parameter k (the number of randomly chosen paths), in particular in symmetric network topologies (like FatTree) small values of k will suffice.

The result presented in this chapter is based on papers [49, 50].

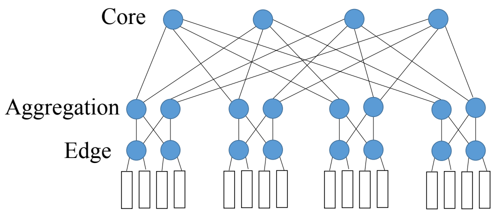
2.1.3 Notations

Given a sequence of random variables $\{X_n\}$, $X_n \Rightarrow X$ indicates convergence in distribution, and $X_n \rightarrow X$ indicates the almost sure convergence. Given a Markov process $\{X(t)\}$, $X(\infty)$ denotes a random variable whose distribution is the same as the steady-state distribution of $X(t)$ (when it exists). $\|\cdot\|$ is the Euclidian norm in $\mathbb{R}^n$. $d(x, S) = \min_{s \in S} \|s - x\|$ is the distance of x from the set S. 'u.o.c.' means uniformly over compact sets.

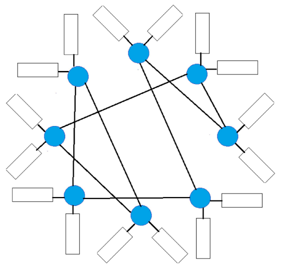
2.2 Model and Problem Statement

2.2.1 data center Network Model

We consider a data center (DC) consisting of a set of servers (host machines) connected by a collection of switches and links. Depending on the DC network topology, all or a subset of the switches are directly connected to servers; for example, in FatTree [2] (Figure 2.1a) only the edge (top-of-the-rack) switches are connected to servers, while in JellyFish [22] (Figure 2.1b) all the switches have some ports connected to servers. Nevertheless, we can model any general DC network topology (FatTree, JellyFish, etc.) by a graph $G(V, E)$ where V is the set of switches and



(a) FatTree.



(b) JellyFish (random graph).

Figure 2.1: Connecting 16 servers (rectangles) using 4-port switches (circles).

E is the set of communication links. A path between two switches is defined as a set of links that connects the switches and does not intersect itself. The paths between the same pair of source-destination switches may intersect with each other or with other paths in DC.

2.2.2 Traffic Model

Each server can generate a flow destined to some other server. We assume that each flow belongs to a set of flow types $\mathcal{J}$. A flow of type $j \in \mathcal{J}$ is a triple (a_j, d_j, s_j) where $a_j \in V$ is its source switch (i.e., the switch connected to the source server), $d_j \in V$ is its destination switch (i.e., the switch connected to its destination server), and s_j is its size (bandwidth requirement). Note that based on this definition, we only need to find the routing of flows in the switch network $G(V, E)$ since the routing from the source server to the source switch or from the destination switch to the destination server is trivial (follows the direct link from the server to the switch). Further, two switches can have *more than one* flow type with different sizes. We assume that type-j flows are generated according to a Poisson process with rate λ_j, and each flow remains in the system for an exponentially distributed amount of time with mean $1/\mu_j$. It is possible to extend our results to a more general model of flow arrival and service time, e.g., when the arrival process is a "renewal" process and service time distribution has lower bounded "hazard rate", using a similar approach as in [51]. We will also report simulation results in Section 2.5 that show that our myopic algorithm indeed performs very well under much more general arrival and service time processes.

For any $j \in \mathcal{J}$, let R_j denote the set of available paths from a_j to d_j, then each type-j flow must be accommodated by using only one of the paths from R_j (i.e., the flow cannot be split among multiple paths). Note that R_j could be the set of all possible paths from a_j to d_j or a subset of them as desired by the network operator. We assume that R_j is nonempty for each $j \in \mathcal{J}$. Define $Y_i^{(j)}(t)$ to be the number of type-j flows routed along the path $i \in R_j$ at time t. The network state is defined as

$$Y(t) = \left(Y_i^{(j)}(t); i \in R_j, j \in \mathcal{J} \right). \tag{2.1}$$

The online (Markov) scheduling algorithm determines the path where an arriving flow at time t is placed, as a function of the current network state $Y(t)$.

We also define $X^{(j)}(t) = \sum_{i \in R_j} Y_i^{(j)}(t)$ which is the total number of type-j flows in the network at time t. Let $Z_l(t)$ be the total amount of traffic (congestion) over link $l \in E$. Based on our notations,

$$Z_l(t) = \sum_{j \in \mathcal{J}} \sum_{i: i \in R_j, l \in i} s_j Y_i^{(j)}(t), \tag{2.2}$$

where by $l \in i$ we mean that link l belongs to path i. We also define $\rho_j = \lambda_j / \mu_j$ which is the mean offered load by type-j flows.

Note that under any Markov scheduling algorithm, the network state $\{Y(t)\}_{t \geq 0}$ is a continuous-time, irreducible Markov chain. It is also positive recurrent, because the total number of type-j flows $X^{(j)}(t)$ in the system is a Markov chain independent of the scheduling algorithm, and its stationary distribution is Poisson with mean ρ_j. Therefore, the process $\{Y(t)\}_{t \geq 0}$ has a unique stationary distribution as $t \to \infty$.

2.2.3 Problem Formulation

For the purpose of load balancing, the network can attempt to optimize different objectives [52] such as minimizing the maximum link congestion in the network or minimizing the sum of link costs where each link cost is a convex function of the link congestion (e.g. this could be a link latency measure [53]). Under both objectives, the traffic needs to be distributed and balanced among the feasible paths in the network, which is essential for maintaining low end-to-end delay for different flows. In this chapter, we use the latter objective but by choosing proper cost functions, an optimal solution to the later objective can be used to also approximate the former objective as we see below.

We define $g(Z_l)$ to be the cost of link l when its congestion is Z_l. Our goal is to find a flow scheduling algorithm that assigns each flow to a single path in the network so as to minimize the mean network cost in the long run, specifically,

$$\begin{aligned} &\text{minimize } \lim_{t \to \infty} \mathbb{E}\left[F(Y(t))\right] \\ &\text{subject to: serving each flow using one path,} \end{aligned} \tag{2.3}$$

where, $F(Y(t)) = \sum_{l \in E} g(Z_l(t))$. We consider polynomial cost functions of the form

$$g(x) = \frac{x^{1+\alpha}}{1+\alpha}, \ \alpha > 0, \tag{2.4}$$

where $\alpha > 0$ is a constant. Thus g is increasing and strictly convex in x. As $\alpha \to \infty$, the optimal solution to (2.3) approaches the optimal solution of the optimization problem whose objective is to minimize the maximum link congestion in the network[2].

2.3 Algorithm Description

In this section, we describe our myopic algorithm for flow assignment where each flow is assigned to one path in the network (no splitting) without interrupting/migrating the ongoing flows in the network. Recall that $Y(t) = (Y_i^{(j)}(t))$ is the network state, $Y_i^{(j)}(t)$ is the number of type-j flows on path $i \in R_j$, and $Z_l(t)$ is the total traffic on link l given by (2.2).

First, we define two forms of *link marginal cost* that measure the increase in the link cost if an

[2]Here we have considered identical links for simplicity but the analysis is easily extendable to the case that $g(\cdot)$ is a function of x/c_l where c_l is the link capacity, or the case that each link has a weight and the goal is to minimize the weighted summation of the link costs.

Algorithm 1 Myopic Flow Scheduling Algorithm

```
Suppose a type-$j$ flow arrives at time $t$ when the system is in state $\mathbf{Y}(t)$. Then,
1: Compute the path marginal costs $w_i^{(j)}(Y(t))$, $i \in R_j$, in either of the forms below:
     • Integral form:
         $$w_i^{(j)}(Y(t)) = \sum_{l \in i} \Delta_l^{(j)}(Y(t)), \quad (2.5)$$
     • Differential form:
         $$w_i^{(j)}(Y(t)) = \sum_{l \in i} \delta_l^{(j)}(Y(t)). \quad (2.6)$$
2: Place the flow on a path $i$ such that
         $$i = \arg\min_{k \in R_j} w_k^{(j)}(Y(t)). \quad (2.7)$$
   Break ties in (2.7) uniformly at random.
```

arriving type-j flow at time t is routed using a path that uses link l.

Definition 1. *(Link marginal cost) For each link l and flow-type j, the link marginal cost is defined in either of the forms below.*

- *Integral form:*

$$\Delta_l^{(j)}(Y(t)) = g\Big(Z_l(t) + s_j\Big) - g\Big(Z_l(t)\Big). \tag{2.8}$$

- *Differential form:*

$$\delta_l^{(j)}(Y(t)) = s_j g'\big(Z_l(t)\big). \tag{2.9}$$

Based on the link marginal costs, we can characterize the increase in the network cost if an arriving type-j flow at time t is routed using path $i \in R_j$. Specifically, let $Y(t^+) = Y(t) + e_i^{(j)}$, where $e_i^{(j)}$ denotes a vector whose corresponding entity to path i and flow type j is one, and its other entities are zero. Then $F(Y(t))$ is the network cost before the type-j flow arrival, and $F(Y(t^+))$ is the network cost after assigning the type-j flow to path i. Then, it is easy to see that

$$\begin{aligned} F(Y(t^+)) - F(Y(t)) &= \sum_{l \in i} \left[g\big(Z_l(t) + s_j\big) - g\big(Z_l(t)\big) \right] \\ &= \sum_{l \in i} \Delta_l^{(j)}(Y(t)). \end{aligned} \tag{2.10}$$

Similarly, based on the differential marginal costs, we have

$$\frac{\partial F(Y(t))}{\partial Y_i^{(j)}(t)} = \sum_{l \in i} s_j g'\big(Z_l(t)\big) = \sum_{l \in i} \delta_l^{(j)}(Y(t)). \tag{2.11}$$

Algorithm 1 describes our myopic flow assignment algorithm that places the newly generated flow on a path that minimizes the increase in the network cost based on either forms (2.10) or (2.11). Upon arrival of a flow, Algorithm 1 takes the corresponding feasible paths and their link congestions into the account for computing the path marginal costs $w_i^{(j)}(t)$ but it does not require to know any information about the other links in the network. The two forms (2.5) and (2.6) are essentially identical in our asymptotic performance analysis in the next section, however it seems slightly easier to work with the differential form (2.6). Algorithm 1 can be implemented either centrally or in a distributed manner using a distributed shortest path algorithm that uses the link marginal costs, $\Delta_l^{(j)}(t)$ or $\delta_l^{(j)}(t)$, as link weights.

Remark 1. *Note that in Algorithm 1 the flow is assigned to a path with the minimum path marginal cost. The path with the minimum path marginal cost is not necessarily the same as the path with the minimum end-to-end congestion (sum of link congestions in the path).*

2.4 Performance Analysis via Fluid Limits

The system state $\{Y(t)\}_{t\geq 0}$ is a stochastic process which is not easy to analyze, therefore we analyze the fluid limits of the system instead. Fluid limits can be interpreted as the first order approximation to the original process $\{Y(t)\}_{t\geq 0}$ and provide valuable qualitative insight into the operation of Algorithm 1. In this section, we introduce the fluid limits of the process $\{Y(t)\}_{t\geq 0}$

and present our main result regarding the convergence of Algorithm 1 to the optimal cost. We deliberately defer the rigorous claims and proofs about the fluid limits to Section 2.7 and for now mainly focus on the convergence analysis to the optimal cost, which is the main contribution of this chapter.

2.4.1 Informal Description of Fluid Limit Process

In order to obtain the fluid limits, we scale the process in rate and space. Specifically, consider a sequence of systems $\{Y^r(t)\}_{t\geq 0}$ indexed by a sequence of positive numbers r, each governed by the same statistical laws as the original system with the flow arrival rates $r\lambda_j$, $j \in \mathcal{J}$ (therefore, a system with a larger r would experience heavier traffic), and initial state $Y^r(0)$ such that $Y^r(0)/r \to y(0)$ as $r \to \infty$ for some fixed $y(0)$. The fluid-scale process is defined as $y^r(t) = Y^r(t)/r$, $t \geq 0$. We also define $y^r(\infty) = Y^r(\infty)/r$, the random state of the fluid-scale process in steady state. If the sequence of processes $\{y^r(t)\}_{t\geq 0}$ converges to a process $\{y(t)\}_{t\geq 0}$ (uniformly over compact time intervals, with probability 1 as $r \to \infty$), the process $\{y(t)\}_{t\geq 0}$ is called the fluid limit. Then, $y_i^{(j)}(t)$ is the fluid limit number of type-j flows routed through path i. Accordingly, we define $z_l^r(t) = Z_l^r(t)/r$ and $x^{(j)^r}(t) = X^{(j)^r}(t)/r$ and their corresponding limits as $z_l(t)$ and $x^{(j)}(t)$ as $r \to \infty$. The fluid limits under Algorithm 1 follow possibly random trajectories, and might not be continuously differentiable; nevertheless, they satisfy the following set of differential equations. We state the result as the following lemma whose proof can be found in Section 2.7.

Lemma 1. *(Fluid equations) Any fluid limit $y(t)$ satisfies the following equations. For any $j \in \mathcal{J}$,* *and* $i \in R_j$,

$$\frac{\mathrm{d}}{\mathrm{d}t} y_i^{(j)}(t) = \lambda_j p_i^{(j)}(y(t)) - \mu_j y_i^{(j)}(t) \tag{2.12a}$$

$$p_i^{(j)}(y(t)) = 0 \textit{ if } i \notin \arg\min_{k \in R_j} w_k^{(j)}(y(t)) \tag{2.12b}$$

$$p_i^{(j)}(y(t)) \geq 0, \sum_{i \in R_j} p_i^{(j)}(y(t)) = 1 \tag{2.12c}$$

$$w_i^{(j)}(y(t)) = \sum_{l \in i} s_j g'(z_l(t)). \tag{2.12d}$$

Equation (2.12a) is simply an accounting identity for $y_i^{(j)}(t)$ stating that, on the fluid-scale, the number of type-j flows over path $i \in R_j$ increases at rate $\lambda_j p_i^{(j)}(y(t))$, and decreases at rate $y_i^{(j)} \mu_j$ due to departures of type-j flows on path i. $p_i^{(j)}(y(t))$ is the fraction of type-j flow arrivals placed on path i. $w_i^{(j)}(y(t))$ is the fluid-limit marginal cost of routing type-j flows in path i when the system is in state $y(t)$. Equation (2.12b) follows from (2.7) and states that the flows can only be placed on the paths which have the minimum marginal cost $\min_{k \in R_j} w_k^{(j)}(y(t))$.

It follows from (2.12a) and (2.12c) that the total number of type-j flows in the system, i.e., $x^{(j)}(t) = \sum_{i \in R_j} y_i^{(j)}(t)$, follows a deterministic trajectory described by the following equation,

$$\frac{\mathrm{d}}{\mathrm{d}t} x^{(j)}(t) = \lambda_j - \mu_j x^{(j)}(t), \ \forall j \in \mathcal{J}, \tag{2.13}$$

which clearly implies that

$$x^{(j)}(t) = \rho_j + (x^{(j)}(0) - \rho_j) e^{-\mu_j t} \ \forall j \in \mathcal{J}. \tag{2.14}$$

Consequently at steady state,

$$x^{(j)}(\infty) = \rho_j, \ \forall j \in \mathcal{J}, \tag{2.15}$$

which means that, in steady state, there is a total of ρ_j type-j flows on the fluid scale.

2.4.2 Main Result and Asymptotic Optimality

In this section, we state our main result regarding the asymptotic optimality of our myopic algorithm. First note that by (2.15), the values of $y(\infty)$ are confined to a convex compact set Υ defined below

$$\Upsilon \equiv \{y = (y_i^{(j)}) : y_i^{(j)} \geq 0, \sum_{i \in R_j} y_i^{(j)} = \rho_j, \ \forall j \in \mathcal{J}\}. \tag{2.16}$$

Consider the problem of minimizing the network cost in steady state on the fluid scale (the counterpart of optimization (2.3)),

$$\begin{aligned} \min \ & F(y) \\ \text{s. t.} \ & y \in \Upsilon \end{aligned} \tag{2.17}$$

Denote by $\Upsilon^\star \subseteq \Upsilon$ the set of optimal solutions to the optimization (2.17). The following proposition states that the fluid limits of Algorithm 1 indeed converge to an optimal solution of the optimization (2.17).

Proposition 1. *Consider the fluid limits of the system under Algorithm 1 with initial condition* $y(0)$*, then as* $t \to \infty$

$$d(y(t), \Upsilon^\star) \to 0. \tag{2.18}$$

Convergence is uniform over initial conditions chosen from a compact set.

The theorem below makes the connection between the fluid limits and the original optimization problem (2.3). It states the main result of this chapter which is the asymptotic optimality of Algorithm 1.

Theorem 1. *Let* $Y^r(t)$ *and* $Y^r_{opt}(t)$ *be respectively the system trajectories under Algorithm 1 and any optimal algorithm for the optimization (2.3). Then in steady state,*

$$\lim_{r \to \infty} \frac{\mathbb{E}\left[F(Y^r(\infty))\right]}{\mathbb{E}\left[F(Y^r_{opt}(\infty))\right]} = 1. \tag{2.19}$$

For example, one optimal algorithm that solves (2.3) is the one that every time a flow arrives or departs, it re-routes the existing flows in the network in order to minimize the network cost at all times. Of course this requires solving a complex combinatorial problem every time a flow arrives/departs and further it interrupts/migrates the existing flows. Under any algorithm (including our myopic algorithm and the optimal one), the mean number of flows in the system in steady state is O(r). Thus by Theorem 1, Algorithm 1 has roughly the same cost as the optimal cost when the number of flows in the system is large, but at much lower complexity and with no migrations/interruptions.

The rest of this section is devoted to the proof of Proposition 1. The proof of Theorem 1 relies on Proposition 1 and is provided in Section 2.7.

2.4.3 Proof of Proposition 1

We first characterize the set of optimal solutions $\Upsilon^\star$ using KKT conditions in the lemma below.

Lemma 2. *Let $\Gamma_j = \{i \in R_j : y_i^{(j)} > 0\} \subseteq R_j,\ j \in \mathcal{J}$. A vector $y \in \Upsilon^\star$ iff $y \in \Upsilon$ and there exists a vector $\eta \geq 0$ such that*

$$w_i^{(j)}(y) = \eta_j,\ \forall i \in \Gamma_j, \tag{2.20a}$$

$$w_i^{(j)}(y) \geq \eta_j,\ \forall i \in R_j \setminus \Gamma_j, \tag{2.20b}$$

where $w_i^{(j)}(\cdot)$ defined in (2.12d).

Proof. Consider the following optimization problem,

$$\min\ F(y) \tag{2.21a}$$

$$\text{s.t.}\ \sum_{i \in R_j} y_i^{(j)} \geq \rho_j,\ \forall j \in \mathcal{J} \tag{2.21b}$$

$$y_i^{(j)} \geq 0,\ \forall j \in \mathcal{J},\ \forall i \in R_j. \tag{2.21c}$$

Since $F(y)$ is an strictly increasing function with respect to $y_i^{(j)}$, for all $j \in \mathcal{J}, i \in R_j$, it is easy to check that the optimization (2.17) has the same set of optimal solutions as the optimization (2.21). Moreover, both optimizations have the same optimal value. Hence we can use the Lagrange multipliers $\eta_j \geq 0$ and $\nu_i^{(j)} \geq 0$ to characterize the Lagrangian as follows.

$$\begin{aligned} L(\eta, \nu, y) =& F(y) + \sum_{j \in \mathcal{J}} \eta_j (\rho_j - \sum_{i; i \in R_j} y_i^{(j)}) \\ & - \sum_{j \in \mathcal{J}} \sum_{i; i \in R_j} \nu_i^{(j)} y_i^{(j)}. \end{aligned} \tag{2.22}$$

From KKT conditions [54], $y \in \Upsilon^\star$, if and only if there exist vectors η and ν such that the following holds.

Feasibility:

$$y \in \Upsilon, \tag{2.23a}$$

$$\eta_j \geq 0,\ \nu_i^{(j)} \geq 0\ \ \forall j \in \mathcal{J},\ i \in R_j, \tag{2.23b}$$

Complementary slackness:

$$\eta_j (\rho_j - \sum_{i; i \in R_j} y_i^{(j)}) = 0,\ \ \forall j \in \mathcal{J}, \tag{2.24a}$$

$$\nu_i^{(j)} y_i^{(j)} = 0,\ \ \forall j \in \mathcal{J},\ i \in R_j, \tag{2.24b}$$

Stationarity:

$$\frac{\partial L(\eta, \nu, y)}{\partial y_i^{(j)}} = 0.\ \ \forall j \in \mathcal{J},\ i \in R_j. \tag{2.25a}$$

Note that (2.23a) implies (2.24a). It follows from (2.25a) that

$$\frac{\partial F(y)}{\partial y_i^{(j)}} = \eta_j + \nu_i^{(j)},\ \forall j \in \mathcal{J}, i \in R_j. \tag{2.26}$$

Define Γ_j as in the statement of the lemma. Note that Γ_j is nonempty for all $j \in \mathcal{J}$ by (2.23a). Then combining (2.24b) and (2.26), $\forall j \in \mathcal{J}$, and noting that $\frac{\partial F(y)}{\partial y_i^{(j)}} = w_i^{(j)}(y)$ by definition, yields (2.20a)-(2.20b). □

Next, we show that the set of optimal solutions $\Upsilon^\star$ is an invariant set of the fluid limits, using the fluid limit equations (2.12a)-(2.12d), and Lemma 2.

Lemma 3. *$\Upsilon^\star$ is an invariant set for the fluid limits, i.e., starting from any initial condition $y(0) \in \Upsilon^\star$, $y(t) \in \Upsilon^\star$ for all $t \geq 0$.*

Proof. Consider a type-j flow and let $I^{(j)}(t) = \arg\min_{i\in R_j} w_i^{(j)}(y(t))$ be the set of paths with the minimum path marginal cost. Note that $\sum_{i\in I^{(j)}(t)} p_i^{(j)}(t) = 1$, $t \geq 0$, by (2.12b), therefore

$$\frac{\mathrm{d}}{\mathrm{d}t}\Big(\sum_{i\in I_i^{(j)}(t)} y_i^{(j)}(t)\Big) = \lambda_j - \Big(\sum_{i\in I^{(j)}(t)} y_i^{(j)}(t)\Big)\mu_j. \tag{2.27}$$

Since $y(0) \in \Upsilon^\star$, it follows from Lemma 2 that $\sum_{i\in I^{(j)}(0)} y_i^{(j)}(0) = \rho_j$. Hence, Equation (2.27) has a unique solution for $\sum_{i\in I^{(j)}(t)} y_i^{(j)}(t)$ which is

$$\sum_{i\in I^{(j)}(t)} y_i^{(j)}(t) = \rho_j,\ t \geq 0. \tag{2.28}$$

On the other hand, since $x^{(j)}(0) = \rho_j$, by (2.14),

$$x^{(j)}(t) = \sum_{i\in R_j} y_i^{(j)}(t) = \rho_j,\ t \geq 0. \tag{2.29}$$

Equations (2.28) and 2.29 imply that, at any time $t \geq 0$, $y_i^{(j)}(t) = 0$ for $i \notin I^{(j)}(t)$, and $y_i^{(j)}(t) \geq 0$ for $i \in I^{(j)}(t)$ such that $\sum_{i\in I^{(j)}(t)} y_i^{(j)}(t) = \rho_j$. Hence, $y(t) = \Big(y_i^{(j)}(t)\Big) \in \Upsilon^\star$ by using $\eta_j(t) = \min_{k\in R_j} w_k^{(j)}(y(t))$ in Lemma 2. □

Next, we show that the fluid limits indeed converge to the invariant set $\Upsilon^\star$ starting from an initial condition in Υ.

Lemma 4. *(Convergence to the invariant set) Consider the fluid limits of the system under Algorithm 1 with initial condition $y(0) \in \Upsilon$, then*

$$d(y(t), \Upsilon^\star) \to 0. \tag{2.30}$$

Also convergence is uniform over the set of initial conditions Υ.

Proof. Starting from $y(0) \in \Upsilon$, (2.14) implies that

$$x^{(j)}(t) = \sum_{i \in R_j} y_i^{(j)}(t) = \rho_j \quad \forall j \in \mathcal{J}, \tag{2.31}$$

at any time $t \geq 0$. To show convergence of $y(t)$ to the set $\Upsilon^\star$, we use a Lyapunov argument. Specifically, we choose $F(.)$ as the Lyapunov function and show that $(\mathrm{d}/\mathrm{d}t)F(y(t)) < 0$ if $y(t) \notin \Upsilon^\star$. Let $\eta_j(y(t)) = \min_{k \in R_j} w_k^{(j)}(y(t))$. Then

$$\begin{aligned}(\mathrm{d}/\mathrm{d}t)F(y(t)) &= \sum_{j \in \mathcal{J}} \sum_{i \in R_j} \frac{\partial F(y)}{\partial y_i^{(j)}} \frac{\mathrm{d}y_i^{(j)}(t)}{\mathrm{d}t} \\ &= \sum_{j \in \mathcal{J}} \mu_j \big[\rho_j \sum_{i \in R_j} w_i^{(j)}(y(t)) p_i^{(j)}(t) - \sum_{i \in R_j} w_i^{(j)}(y(t)) y_i^{(j)}(t)\big] \\ &\overset{(a)}{=} \sum_{j \in \mathcal{J}} \mu_j \big[\rho_j \eta_j(y(t)) - \sum_{i \in R_j} w_i^{(j)}(y(t)) y_i^{(j)}(t)\big] \\ &\overset{(b)}{<} \sum_{j \in \mathcal{J}} \mu_j \big[\rho_j \eta_j(y(t)) - \eta_j(y(t)) \sum_{i \in R_j} y_i^{(j)}(t)\big] \overset{(c)}{=} 0.\end{aligned} \tag{2.32}$$

Equality (a) follows from the fact that $p_i^{(j)}(t) = 0$ if $w_i^{(j)}(t) > \eta_j(t)$, and $\sum_{i \in I^{(j)}(t)} p_i^{(j)}(t) = 1$, $t \geq 0$, by (2.12b) and (2.12c). Inequality (b) follows from the fact that $y(t) \notin \Upsilon^\star$, so by Lemma 2, there exists an $i \in R_j$ such that $y_i^{(j)}(t) > 0$ but $w_i^{(j)}(y(t)) > \eta_j(y(t))$. Equality (c) holds because of (2.31). □

Now we are ready to complete the proof of Proposition 1, i.e., to show that starting from any initial condition in a compact set, uniform convergence to the invariant set $\Upsilon^\star$ holds.

Proof of Proposition 1. First note that $(\mathrm{d}/\mathrm{d}t)F(y(t))$ (as given by (2.32)) is a continuous function with respect to $y(t) = (y_i^{(j)}(t) \geq 0)$. This is because the path marginal costs $w_i^{(j)}(y(t))$ are continuous functions of $y(t)$ and so is their minimum $\eta_j(y(t)) = \min_{i \in R_j} w_i^{(j)}(y(t))$.

Next, note that by Lemma 4, for any $\epsilon_1 > 0$, and $a \in \Upsilon$, there exists an $\epsilon_2 > 0$ such that if $F(a) - F(\Upsilon^\star) \geq \epsilon_1$ then,

$$(\mathrm{d}/\mathrm{d}t)F(y(t))\big|_{y(t)=a} \leq -\epsilon_2 \tag{2.33}$$

By the continuity of $(\mathrm{d}/\mathrm{d}t)F(y(t))$ in $y(t)$, there exists a $\delta > 0$ such that $\|y(t) - a\| \leq \delta$ implies,

$$|(\mathrm{d}/\mathrm{d}t)F(y(t)) - (d/dt)F(a)| \leq \epsilon_2/2 \tag{2.34}$$

Combining (2.33) and (2.34), for all $y(t)$ such that $\|y(t) - a\| \leq \delta$,

$$(\mathrm{d}/\mathrm{d}t)F(y(t)) \leq -\epsilon_2/2.$$

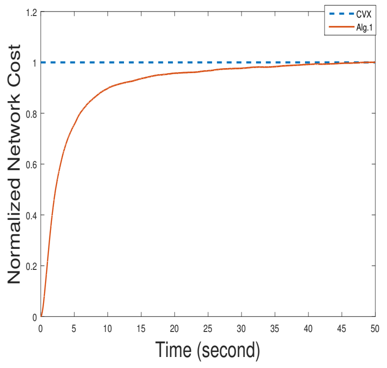
By (2.14), for any $\delta > 0$, we can find t_δ large enough such that for all $t > t_\delta$, $\|y(t) - a\| \leq \delta$ for some $a \in \Upsilon$.

Putting everything together, for any $\epsilon_1 > 0$, there exists $\epsilon_2 > 0$ such that if $F(y(t)) - F(\Upsilon^\star) \geq \epsilon_1$ then $(\mathrm{d}/\mathrm{d}t)F(y(t)) \leq -\epsilon_2/2 < 0$. Applying Lyapunov argument with $F(.)$ as Lyapunov function completes the proof of Proposition 1. □

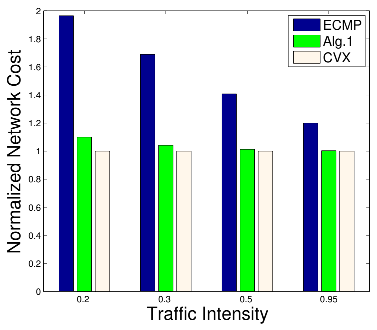
2.5 Simulation Results

In this section, we provide simulation results and evaluate the performance of Algorithm 1 under a wide range of traffic conditions in the following data center architectures:

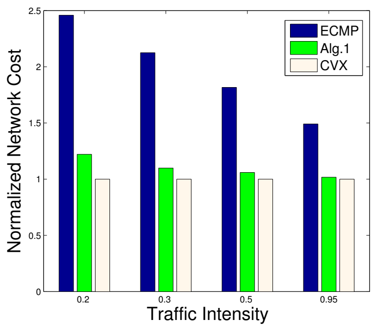
- *FatTree* which consists of a collection of edge, aggregation, and core switches and offers equal length path between the edge switches. Figure 2.1a shows a FatTree with 16 servers and 8 4-port edge switches. For simulations, we consider a FatTree with 128 servers and 32 8-port edge switches.



(a) Convergence of network cost



(b) Exponential traffic model



(c) Empirical traffic model

Figure 2.2: Experimental Results for FatTree. **(a)**: Convergence of the network cost under Algorithm 1, normalized with the lower-bound on the optimal solution (CVX), to 1. The scaling parameter r is 100 here. **(b)** and **(c)**: Performance ratio of Algorithm 1 and ECMP in FatTree, normalized with the lower-bound (CVX) for exponential and empirical traffic models.

- *JellyFish* which is a random graph in which each switch i has k_i ports out of which r_i ports are used for connection to other switches and the remaining $k_i - r_i$ ports are used for connection to servers. Figure 2.1b shows a JellyFish with 4-port switches, and $k_i = 4$, $r_i = 2$ for all the switches. For simulations, we consider a JellyFish constructed using 20 8-port switches and 100 servers. Each 8-port switch is connected to 5 servers and 3 remaining links are randomly connected to other switches (this corresponds to $k_i = 8$, $r_i = 3$ for all the switches).

For the 128-server FatTree, when source and destination switches are located in different (same) racks, our myopic algorithm considers 16 (4) equal length candidate paths. For the case of d-regular random graphs (where each node has d edges), the number of paths between 2 switches can be very large which could significantly increase the computational complexity of the algorithm. To reduce the computation overhead, we can neglect the long paths since such paths will naturally have large marginal costs and will not be used by Algorithm 1. In our simulations, for the case of JellyFish, we consider (at most) the first 20 shortest paths (in terms of the number of links) for each pairs of switches.

Our rationale for selecting these architectures stems from the fact that they are on two opposing sides of the spectrum of topologies: while FatTree is a highly structured topology, JellyFish is a random topology; hence they should provide a good estimate for the robustness of Algorithm 1 to different network topologies and possible link failures.

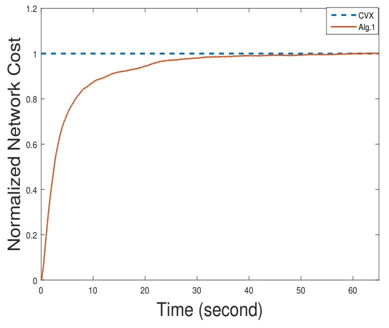
We generate the flows under two different traffic models to which we refer to as *exponential model* and *empirical model*:

- *Exponential model*: Flows are generated per Poisson processes and exponentially distributed durations. The parameters of duration distribution is chosen uniformly at random from 0.5 to 1.5 for different flows to simulate a more dynamic range of flow durations. The flow sizes are chosen according to a log-normal distribution.
- *Empirical model*: Flows are generated based on recent empirical studies on characterization of data center traffic. As suggested by these studies, we consider log-normal inter-arrival times [55], service times based on the empirical result in [23], and log-normal flow sizes [55]. Particularly, the most periods of congestion tend to be short lived, namely, more than 90% of the flows that are more than 1 second long, are no longer than 2 seconds [23].

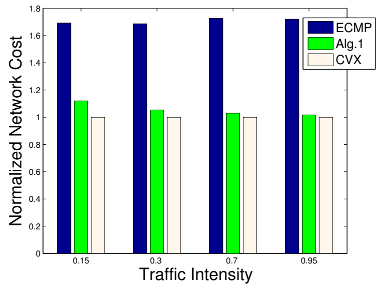
In both models, the flow sizes are log-normal with mean 1.2 and standard deviation 0.4. This generates flow sizes ranging from 1% to 40% of link capacity with high probability to capture the nature of flow sizes in terms of “mice” and “elephant” flows. Furthermore, we consider a random traffic pattern, i.e., source and destination of flows are chosen uniformly at random. The link cost parameter α is chosen to be 1 in these simulations.

Under both models, to change the traffic intensity, we keep the other parameters fixed and scale the arrival rates (with parameter r).

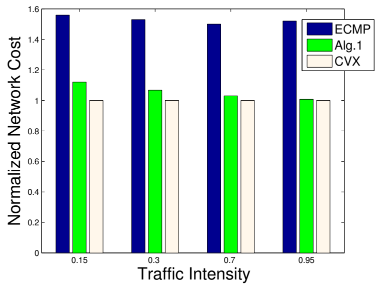
We report the simulation results in terms of the performance ratio between Algorithm 1 and a benchmark algorithm (similar to (2.19)). Since the optimal algorithm (e.g. the one that every time a flow arrives or departs, it re-routes the existing flows in the network in order to minimize the network cost at all times) is hard to implement (and even unknown), instead we use a convex relaxation method to find a lower-bound on the optimal cost at each time. We note that, for Fat-Tree topology, equal splitting of every flow among its candidate paths is optimal. For JellyFish topology, every time a flow arrives or departs, we use CVX [56], to minimize $F(Y(t))$, by relaxing



(a) Convergence of network cost



(b) Exponential traffic model



(c) Empirical traffic model

Figure 2.3: Experimental Results for JellyFish. **(a)**: Convergence of the network cost under Algorithm 1 in JellyFish, normalized with the lower-bound on the optimal solution (CVX), to 1. The scaling parameter r is 100 here. **(b)** and **(c)**: Performance ratio of Algorithm 1 and ECMP in JellyFish, normalized with the lower-bound (CVX) for exponential and empirical traffic models.

the combinatorial constraints, i.e., allowing splitting of flows among multiple paths and re-routing the existing flows. We compare the network cost under Algorithm 1 and traditional ECMP (which statically assigns flows to the shortest paths (in number of links) via flow hashing.), normalized by the lower-bound on the optimal solution (to which we refer to as CVX in the plots).

2.5.1 Experimental Results for FatTree

Figure 2.2a shows that the aggregate cost under Algorithm 1 indeed converges to the optimal solution (normalized cost ratio goes to 1) which verifies Theorem 1. Figures 2.2b and 2.2c show the cost performance under Algorithm1 and ECMP, normalized by the CVX lower-bound, under the exponential and the empirical traffic models respectively. The traffic intensity is measured in terms of the ratio between the steady state offered load and the bisection bandwidth. For FatTree, the bisection bandwidth depends on the number of core switches and their number of ports. As we can see, our myopic algorithm is very close to the lower-bound on the optimal value (CVX) for light, medium, and high traffic intensities. As it is shown, the performance improves at higher traffic intensities which correspond to larger values of r in Theorem 1. They also suggest that Theorem 1 holds under more general arrival and service time processes. In this simulations, Algorithm 1 gave a performance improvement ranging form 50% to more than 100%, compared to ECMP, depending on the traffic intensity, under the empirical traffic model. The standard deviation (SD) of performance ratio for 30 different runs ranges from 0.14 to 0.01 for Algorithm 1, and from 0.3 to 0.03 for ECMP as traffic intensity grows.

2.5.2 Experimental Results for JellyFish

Figure 2.3a shows that the aggregate cost under Algorithm 1 indeed converges to the optimal solution which again verifies Theorem 1. Figures 2.3b and 2.3c compare the performance of Algorithm 1 and ECMP, normalized with the lower-boud on the optimal solution (CVX), under both the exponential and empirical traffic models. As before, the traffic intensity is measured by the ratio between the steady state offered load and the bisection bandwidth. To determine the bisection bandwidth, we have used the bounds reported in [57, 58] for regular random graphs. Again we see that our myopic algorithm performs very well in all light, medium, and high traffics. In JellyFish, Algorithm 1 yields performance gains ranging from 60% to 70%, compared to ECMP, under the empirical traffic model. Corresponding SD for 30 different runs ranges from 0.04 to 0.01 for Algorithm 1, and from 0.1 to 0.05 for ECMP as traffic intensity grows.

2.6 Randomized Myopic Algorithms

Algorithm 1 needs to consider all the available paths for an arriving flow and finds the shortest path based on the (integral (2.5) or differential (2.6)) marginal cost of paths. In this section, we describe and empirically evaluate randomized versions of our myopic algorithm which have less complexity than Algorithm 1, while can effectively provide a large fraction of the performance gain obtained by Algorithm 1. Our approach is motivated by the literature on *randomized load balancing* for scheduling jobs in servers, where a widely used idea is that, instead of considering all the servers and assigning the arriving job to the least-loaded server, k servers are first chosen at random (for some $k \geq 2$) and then the job is assigned to the least-loaded server among them. This idea was originally proposed in [6], where it was shown that having $k = 2$ leads to exponential improvement in the expected time a job spends in the system over $k = 1$ which is basically the totally random assignment.

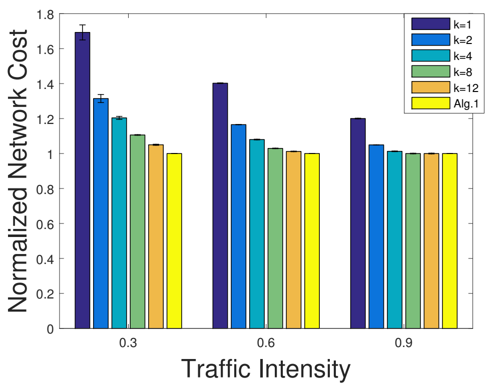
In our setting, a counterpart of this approach can be used for scheduling of flows in paths as follows. Fix k, when a flow is generated, the algorithm chooses k paths at random out of the available paths for the flow, then calculates the marginal costs of these k paths according to the integral or the differential form formulas, and assigns the flow to the path with the minimum path marginal cost among these k paths. See Algorithm 2 for the full description.

Algorithm 2 Randomized Myopic Algorithm with Parameter k

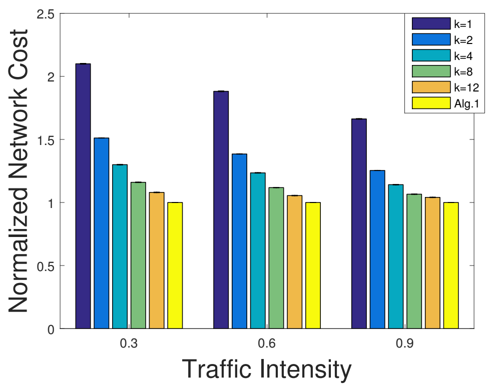
```
Suppose a type-$j$ flow arrives at time $t$ when the system is in state $\mathbf{Y}(t)$. Then,
1: Choose $k$ paths from the set $|R_j|$, uniformly at random, let $R_j^{(k)}$ denotes this subset of paths.
2: Compute the path marginal costs $w_i^{(j)}(Y(t))$, $i \in R_j^{(k)}$, in either of the forms below:
     • Integral form:
         $$w_i^{(j)}(Y(t)) = \sum_{l \in i} \Delta_l^{(j)}(Y(t)),  \qquad (2.35)$$
     • Differential form:
         $$w_i^{(j)}(Y(t)) = \sum_{l \in i} \delta_l^{(j)}(Y(t)).  \qquad (2.36)$$
3: Place the flow on a path $i$ such that
         $$i = \arg\min_{k \in R_j^{(k)}} w_k^{(j)}(Y(t)).  \qquad (2.37)$$
   Break ties in (2.37) uniformly at random.
```

We notice that ECMP in structured topologies like FatTree, where all candidate paths for an arriving flow have the same number of links (same length), is basically the random assignment of flows to the paths which is identical to setting $k = 1$ in Algorithm 2.

Next, we empirically evaluate the performance of Algorithm 2 for different values of k. We present the results for two different topologies and two traffic model as in Section 2.5. For Jelly-Fish, we consider (at most) the first 20 shortest paths (in terms of the number of links) for each pairs of switches to be consistent with Section 2.5.



(a) Exponential traffic model.



(b) Empirical traffic model.

Figure 2.4: Performance of Algorithm 2 with different values of k, in FatTree, normalized with the Algorithm 1.

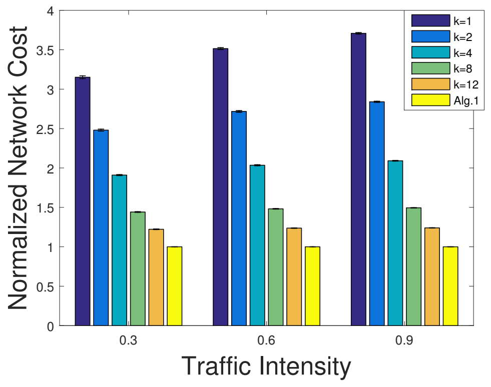
2.6.1 Experimental Results for FatTree

Figures 2.4a and 2.4b show the cost performance under Algorithm 2 with different values of k, normalized by the cost of Algorithm 1, under the exponential and the empirical traffic models respectively. Note that Algorithm 2 with $k = 16$ is equivalent to Algorithm 1, as there are at most 16 available paths for an arriving flow in the FatTree topology we described in Section 2.5. Error bars in all plots correspond to standard deviation of normalized mean network cost computed from results of 30 runs.

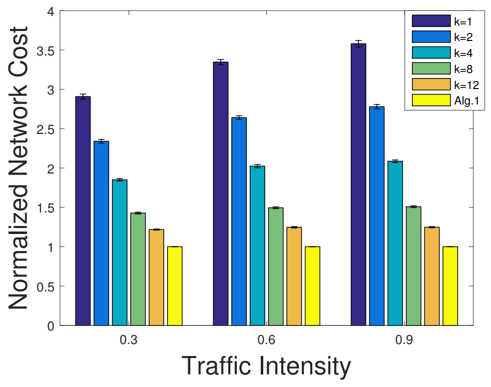
In these two plots, we can see that the maximum improvement in network cost we get by increasing k happens at $k = 2$ compared with random assignment of flows, $k = 1$. Furthermore, as we increase value of k we get smaller improvement in performance. For instance, normalized cost improves about 0.4 by increasing k from 1 to 2, while the improvement from $k = 2$ to $k = 4$ is about 0.1, for traffic intensity equal to 0.3 under exponential model (Figure 2.4a). This behavior is seen in both figures, and is more profound for higher traffic intensity.

2.6.2 Experimental Results for JellyFish

Figures 2.5a and 2.5b show the network cost under Algorithm 2 with different values of k, normalized by the cost of Algorithm 1, under the exponential and the empirical traffic models



(a) Exponential traffic model



(b) Empirical traffic model

Figure 2.5: Performance of Algorithm 2 with different values of k in JellyFish, normalized with the Algorithm 1.

respectively. Note that Algorithm 2 with $k = 20$ is equivalent to Algorithm 1, as there are at most 20 available paths considered between any two switches in the JellyFish topology we described in Section 2.5.

In these figures, we observe the same behavior as what discussed for FatTree: the performance improvement obtained by increasing k by one is larger for smaller k. Also, comparing Figures 2.5a and 2.5b with Figures 2.3b and 2.3c, in order for Algorithm 2 to beat ECMP–which only considers shortest paths (in the terms of the number of links)–we need to choose $k \geq 12$.

We also note that in JellyFish, for small k (e.g., $k = 1, 2$), the normalized cost under the randomized algorithm increases as traffic intensity grows, unlike the results for FatTree. This can be justified by noting that the symmetric structure of FatTree allows random assignment of flows to balance the load better as traffic intensity increases (higher flow arrival rates) because the number of flow-to-path assignment decisions increases. However, in JellyFish the structure is asymmetric and long paths are used more frequently by the randomized algorithm as traffic intensity increases. As a result, the convexity of the link cost function, and the fact that the network cost is the summation of all links' costs, will cause a larger network cost in higher traffic intensities.

Based on the simulations, we conclude that to get a reasonably good performance, we need smaller values of k in FatTree compared to JellyFish. This can be attributed to the fact that all the candidate paths for a flow in the FatTree topology have the same number of links, while in the JellyFish topology, paths can be very different in terms of their number of links. So selection of k paths completely at random, as used in Algorithm 2, might lead to using long paths which contribute more to the network cost. Thus, uniform sampling seems more suitable for symmetric topologies like FatTree.

2.7 Formal Proofs of Fluid Limits and Theorem 1

2.7.1 Proof of Fluid Limits

We prove the existence of fluid limits under Algorithm 1 and derive the corresponding fluid equations (2.12a)-(2.12d). Arguments in this section are quite standard [51], [59], [60]. Recall that $Y^r(t)$ is the system state with the flow arrival rate $r\lambda_j$, $j \in \mathcal{J}$, and initial state $Y^r(0)$. The fluid-scale process is $y^r(t) = Y^r(t)/r$, $t \in [0, \infty)$. Similarly, $z_l^r(t) = Z_l^r(t)/r$ and $x^{(j)^r}(t) = X^{(j)^r}(t)/r$ are defined. We assume that $y^r(0) \to y(0)$ as $r \to \infty$ for some fixed $y(0)$.

We first show that, under Algorithm 1, the limit of the process $\{y^r(t)\}_{t\geq 0}$ exists along a subsequence of r as we show next. The process $Y^r(t)$ can be constructed as follows

$$\begin{aligned} Y_i^{(j)^r}(t) =& Y_i^{(j)^r}(0) + \Pi_{i,j}^a\Big(\int_0^t P_i^{(j)}(Y^r(s)) r\lambda_j ds\Big) \\ & - \Pi_{i,j}^d\Big(\int_0^t \mu_j Y_i^{(j)^r}(s) ds\Big), \ \forall j \in \mathcal{J}, i \in R_j \end{aligned} \tag{2.38}$$

where $\Pi_{i,j}^a(.)$ and $\Pi_{i,j}^d(.)$ are independent unit-rate Poisson processes, and $P_i^{(j)}(Y^r(t))$ is the probability of assigning a type-j flow to path i when the system state is $Y^r(t)$. Note that by the Functional Strong Law of Large Numbers [61], almost surely,

$$\frac{1}{r}\Pi_{i,j}^a(rt) \to t, \text{ u.o.c.}; \ \frac{1}{r}\Pi_{i,j}^d(rt) \to t, \text{ u.o.c.} \tag{2.39}$$

where u.o.c. means uniformly over compact time intervals. Define the fluid-scale arrival and departure processes as

$$\begin{aligned} a^r_{i,j}(t) &= \frac{1}{r}\Pi^a_{i,j}\Big(\int_0^t P^{(j)}_i(Y^r(s))r\lambda_j ds\Big),\\ d^r_{i,j}(t) &= \frac{1}{r}\Pi^d_{i,j}\Big(\int_0^t \mu_j Y^{(j)^r}_i(s)ds\Big). \end{aligned} \tag{2.40}$$

Lemma 5. *(Convergence to fluid limit sample paths) If $y^r(0) \to y(0)$, then almost surely, every subsequence $(y^{r_n}, a^{r_n}, d^{r_n})$ has a further subsequence $(y^{r_{n_k}}, a^{r_{n_k}}, d^{r_{n_k}})$ such that $(y^{r_{n_k}}, a^{r_{n_k}}, d^{r_{n_k}}) \to (y, a, d)$. The sample paths y, a, d are Lipschitz continuous and the convergence is u.o.c.*

Proof. The proof is standard and follows from the fact that $a^r_{i,j}(.)$ and $d^r_{i,j}(.)$ are asymptotically Lipschitz continuous (see e.g., [51], [59], [62] for similar arguments), namely, there exists a constant $C > 0$ such that for $0 \le t_1 \le t_2 < \infty$,

$$\limsup_r (a^r_{i,j}(t_2) - a^r_{i,j}(t_1)) \le C(t_2 - t_1), \tag{2.41}$$

and similarly for $d^r_{i,j}(.)$. More precisely, for arrival process $a^r_{i,j}(.)$, we argue that,

$$\begin{aligned} &\limsup_r (a^r_{i,j}(t_2) - a^r_{i,j}(t_1))\\ &= \limsup_r \frac{1}{r}\Pi^a_{i,j}\Big(\int_{t_1}^{t_2} P^{(j)}_i(Y^r(s))r\lambda_j ds\Big)\\ &\overset{(a)}{\le} \limsup_r \frac{1}{r}\Pi^a_{i,j}\Big(\int_{t_1}^{t_2} r\lambda_j ds\Big)\\ &= \limsup_r \Big(\frac{1}{r}\Pi^a_{i,j}(r\lambda_j(t_2 - t_1))\Big) \end{aligned}$$

where inequality (a) follows from the fact that $P^{(j)}_i(Y^r(s)) \le 1$. Using (2.39), we obtain (2.41). The argument is similar for $d^r_{i,j}(.)$, noting that $(y^r(.))$ is uniformly bounded over any finite time interval for large r. So the limit (y, a, d) exists along the subsequence. □

Proof of Lemma 1. It follows from (2.38), (2.40), (2.39), and the existence of the fluid limits (Lemma 5), that

$$y^{(j)}_i(t) = y^{(j)}_i(0) + a^{(j)}_i(t) - d^{(j)}_i(t),$$

where $d_i^{(j)}(t) = \int_0^t y_i^{(j)}(s)\mu_j ds$, and $\sum_{i\in R_j} a_i^{(j)}(t) = \lambda_j t,\ a_i^{(j)}(t)$ is nondecreasing. The fluid equations (2.12a) and (2.12c) are the diffrential form of these equations (the fluid sample paths are Lipschitz continuous so the derivatives exist almost everywhere), where

$$p_i^{(j)}(t) := \frac{1}{\lambda_j}\frac{\mathrm{d}a_i^{(j)}(t)}{\mathrm{d}t}. \tag{2.42}$$

For any type j, and for $w_i^{(j)}(y(t))$ defined in (2.12d), let

$$w_j^\star(y(t)) = \min_{i\in R_j} w_i^{(j)}(y(t)).$$

Consider any regular time t and a path $i \notin \arg\min_{i\in R_j} w_i^{(j)}(y(t))$. By the continuity of $w_i^{(j)}(y(t))$, there must exist a small time interval (t_1, t_2) containing t such that

$$w_i^{(j)}(y(\tau)) > w_j^\star(\tau)\ \ \forall \tau \in (t_1, t_2).$$

Consequently, for all r large enough along the subsequence,

$$w_i^{(j)}(y^r(\tau)) > w_j^\star(y^r(\tau))\ \ \forall \tau \in (t_1, t_2).$$

Multiplying both sides by r^α, it follows that

$$w_i^{(j)}(Y^r(\tau)) > w_j^\star(Y^r(\tau)),\ \ \forall \tau \in (t_1, t_2).$$

Hence $P_i^{(j)}(Y^r(\tau)) = 0$, $\tau \in (t_1, t_2)$, and $a_i^{r(j)}(t_1, t_2) = 0$, for all r large enough along the subsequence. Therefore $a_i^{(j)}(t_1, t_2) = 0$ which shows that $(\mathrm{d}/\mathrm{d}t)a_i^{(j)}(t) = 0$ at $t \in (t_1, t_2)$. This establishes (2.12b). □

2.7.2 Proof of Theorem 1

We first show that

$$F(y^r(\infty)) \Longrightarrow F^\star, \tag{2.43}$$

where $F^\star = F(\Upsilon^\star)$ is the optimal cost. By Proposition 1 and the continuity of $F(\cdot)$, for any fluid sample path $y(t)$ with initial condition $y(0)$, we can choose t_{ϵ_1} large enough such that given any small $\epsilon_1 > 0$, $|F(y(t_{\epsilon_1})) - F^\star| \leq \epsilon_1$. With probability 1, every subsequence y^{r_n} has a further subsequence $y^{r_{n_k}}$ such that $y^{r_{n_k}}(t) \to y(t)$ u.o.c. (see Lemma 5), hence, by the continuous mapping theorem [61], we also have $F(y^{r_{n_k}}(t)) \to F(y(t))$, u.o.c. For any $\epsilon_2 > 0$, for r_{n_k} large enough, we can choose an $\epsilon_3 > 0$ such that, uniformly over all initial states $y^{r_{n_k}}(0)$ such that $\|y^{r_{n_k}}(0) - y(0)\| \leq \epsilon_3$,

$$\mathbb{P}\{|F(y^{r_{n_k}}(t_{\epsilon_1}) - F(y(t_{\epsilon_1}))| < \epsilon_1\} > 1 - \epsilon_2 \tag{2.44}$$

This claim is true, since otherwise for a sequence of initial states $y^{r_{n_k}}(0) \to y(0)$ we have

$$\mathbb{P}\{|F(y^{r_{n_k}}(t_{\epsilon_1}) - F(y(t_{\epsilon_1}))| < \epsilon_1\} \leq 1 - \epsilon_2,$$

which is impossible because, almost surely, we can choose a subsequence of r_{n_k} along which uniform convergence $F(y^{r_{n_k}}(t)) \to F(y(t))$, with initial condition $y(0)$ holds. Hence,

$$\begin{aligned}
&\mathbb{P}\{|F(y^{r_{n_k}}(t_{\epsilon_1})) - F^\star| < 2\epsilon_1\} \\
&\geq \mathbb{P}\{|F(y^{r_{n_k}}(t_{\epsilon_1}) - F(y(t_{\epsilon_1}))| + |F(y(t_{\epsilon_1})) - F^\star| < 2\epsilon_1\} \\
&\geq \mathbb{P}\{|F(y^{r_{n_k}}(t_{\epsilon_1}) - F(y(t_{\epsilon_1}))| < \epsilon_1\} > 1 - \epsilon_2
\end{aligned}$$

which in particular implies

$$F(y^{r_{n_k}}(\infty)) \Longrightarrow F^\star,$$

because ϵ_1 and ϵ_2 can be made arbitrarily small. Hence, we have shown that every sequence $F(y^{r_n}(\infty))$ has a further subsequence $F(y^{r_{n_k}}(\infty))$ that converges to the same limit $F^\star$ (the unique optimal cost). Therefore in view of Theorem 2.6 of [61], we can conclude that $F(y^r(\infty)) \Longrightarrow F^\star$.

Next, we show (2.19). Under any algorithm (including Algorithm 1 and the optimal one),

$$\sum_{i \in R_j} Y_i^{(j)^r}(\infty)/r = X^{(j)^r}(\infty)/r,$$

where $X^{(j)^r}(\infty)$ has Poisson distribution with mean $r\rho_j$, and $X^{(j)^r}(\infty)$, $j \in \mathcal{J}$, are independent. Let,

$$\bar{s} = \max_{j \in \mathcal{J}} s_j < \infty.$$

The traffic over each link l is clearly bounded as

$$Z_l^r/r < \bar{s} \sum_j X^{(j)^r}(\infty)/r = \bar{s} X^r(\infty)/r,$$

where $X^r(\infty)$ has Poisson distribution with mean $r\sum_j \rho_j$. Hence, $F(y^r(\infty))$ is stochastically dominated by $|E| g\big(\bar{s}X^r(\infty)/r\big)$, and g is polynomial. It then follows that the sequence of random variables $\{F(y^r(\infty))\}$ (and also $\{y^r(\infty)\}$) are uniformly integrable under any algorithm. Then, in view of (2.43), by Theorem 3.5 of [61], under our Algorithm 1.

$$\mathbb{E}\Big[F(Y^r(\infty)/r)\Big] \to F^\star. \tag{2.45}$$

Now consider any optimal algorithm for the optimization (2.3). It holds that

$$F(\mathbb{E}\Big[y_{\text{opt}}^r(\infty)\Big]) \le \mathbb{E}\Big[F(y_{\text{opt}}^r(\infty))\Big] \le \mathbb{E}\Big[F(y^r(\infty))\Big],$$

where the first inequality is by Jensen's inequality, and the second follows from definition of optimality. Taking the limit as $r \to \infty$, it follows by an squeeze argument that

$$\mathbb{E}\Big[F(Y_{\text{opt}}^r(\infty)/r)\Big] \to F^\star. \tag{2.46}$$

Finally, (2.45) and (2.46) will imply (2.19) in view of the polynomial structure of F.

Chapter 3: Coflow Scheduling to Minimize The Weighted Average Completion Time

3.1 Introduction

Many data-parallel computing applications (e.g. MapReduce [4], Hadoop [63, 64], Dryad [5], etc.) consist of multiple computation and communication stages or have machines grouped by functionality. While computation involves local operations in servers, communication takes place at the level of machine groups and involves transfer of many pieces of intermediate data (flows) across groups of machines for further processing. In such applications, the collective effect of all the flows between the two machine groups is more important than that of any of the individual flows. A computation stage often cannot start unless all the required data pieces from the preceding stage are received, or the application latency is determined by the transfer of the last flow between the groups [8, 65].

As an example, consider a MapReduce application. Each mapper performs local computations and writes intermediate data to the disk, then each reducer pulls intermediate data from different mappers, merges them, and computes its output. The job will not finish until its last reducer is completed. Consequently, the job completion time depends on the time that the last flow of the communication phase (called shuffle) is finished. Such intermediate communication stages in a data-parallel application can account on average for about 56% of the job's runtime (see Appendix A in [66] for more detail), and hence can have a significant impact on application performance. Optimizing flow-level performance metrics (e.g. the average flow completion time) have been extensively studied before from both networking systems and theoretical perspective (see, e.g., [48, 34, 49] and references there.), however, these metrics ignore the dependence among the flows of an application which is critical for the application-level performance in data-parallel computing applications.

Recently Chowdhury and Stoica [7] have introduced the *coflow* abstraction to capture these communication patters. *A coflow is defined as a collection of parallel flows whose completion time is determined by the completion time of the last flow in the collection.* Coflows can represent most communication patterns between successive computation stages of data-parallel applications [8]. Clearly the traditional flow communication is still a coflow with a single flow. Jobs from one or more data-parallel applications create multiple coflows in a shared data center network. These coflows could vary widely in terms of the total size of the parallel flows, the number of the parallel flows, and the size of the individual flows in the coflows (e.g., see the analysis of production traces in [8]). Classical flow/job scheduling algorithms do not perform well in this environment [8] because each coflow consists of multiple flows– whose completion time is dominated by its slowest flow– and further, the progress of each flow depends on its assigned rate at *both* its source and its destination. This coupling of rate assignments between the flows in a coflow and across the source-destination pairs in the network is what makes the coflow scheduling problem considerably harder than the classical flow/job scheduling problems.

In this chapter, we study the coflow scheduling problem, namely, the algorithmic task of determining when to start serving each flow and at what rate, in order to minimize the weighted sum of completion times of coflows in the system. In the case of equal weights, this is equivalent to minimizing the average completion time of coflows.

3.1.1 Related Work

Several scheduling heuristics have been already proposed in the literature for scheduling coflows, e.g. [65, 8, 67, 10]. A FIFO-based solution was proposed in [65] which also uses multiplexing of coflows to avoid starvation of small flows which are blocked by large head-of-line flows. A Smallest-Effective-Bottleneck-First heuristic was ierriorntroduced in Varys [8]: it sorts the coflows in an ascending order in a list based on their maximum loads on the servers, and then assigns rates to the flows of the first coflow in the list such that all its flows finish at the same time. The remaining capacity is distributed among the rest of the coflows in the list in a similar fashion to avoid under-utilization of the network. Similar heuristics without prior knowledge of coflows were introduced in Aalo [10]. A joint scheduling and routing of coflows in data center networks was introduced in [67] where similar heuristics based on a Minimum-Remaining-Time-First policy are developed. In a more recent work [68], a randomized algorithm with theoretical guarantee is proposed for coflow scheduling problem when flows can be transmitted at arbitrary small granularity in a general graphs (i.e., rate allocation model, see Section 3.2). This model considers two cases. In the first case, the flow can completely split over multiple paths between its source and destination in the network. In the second model, a single path is specified by the model for each flow along which the flow is transmitted upon scheduling.

Here, we would like to highlight three papers [69, 70, 71] that are more relevant to our work. These papers consider the problem of minimizing the total weighted completion time of coflows with release dates (i.e., coflows arrive over time.) and provide algorithms with provable guarantees. This problem is shown to be NP-complete through its connection with the concurrent open shop problem [8, 69], and then approximation algorithms are proposed which run in polynomial time and return a solution whose value is guaranteed to be within a constant fraction of the optimal (a.k.a., *approximation ratio*). These papers rely on linear programming relaxation techniques from combinatorial scheduling literature (see, e.g., [72, 73, 74]). In [69], the authors utilize an interval-indexed linear program formulation which helps partitioning the coflows into disjoint groups. All coflows that fall into one partition are then viewed as a single coflow, where a polynomial-time algorithm is used to optimize its completion time. Authors in [70] have recently constructed an instance of the concurrent open shop problem (see [75] for the problem definition) from the original coflow scheduling problem. Then applying the well-known approximation algorithms for the concurrent open shop problem to the constructed instance, an ordering of coflows is obtained which is then used in a similar fashion as in [69] to obtain an approximation algorithm. The deterministic algorithm in [70] has better approximation ratio compared to [69], for both cases of with and without release dates. In [71], a linear program approach based on ordering variables is utilized to develop two algorithms, one deterministic and the other randomized. The deterministic algorithm gives the same bounds as in [70], while the randomized algorithm has better performance approximation ratios compared to [69, 70], for both cases of with and without release dates.

3.1.2 Main Contributions

In this chapter, we consider the problem of minimizing the total weighted coflow completion time. Our main contributions can be summarized as follows.

• **Coflow Scheduling Algorithm.** We use a Linear Program (LP) approach based on ordering variables followed by a simple list scheduling policy to develop a deterministic algorithm. Our approach improves the prior algorithms in both cases of with and without release dates. Even if we consider equal weights for all coflows (i.e., minimizing the average completion time), our algorithm has the best known performance guarantee. Table 3.1 summarizes our results in comparison with the prior best-known performance bounds. Performance of a deterministic (randomized) algorithm is defined based on approximation ratio, i.e., the ratio between the (expected) weighted sum of coflow completion times obtained by the algorithm and the optimal value. When coflows have release dates (which is often the case in practice as coflows are generated at different times), our deterministic algorithm improves the approximation ratio of 12 [70, 71] to 5, which is also better than the best known randomized algorithm proposed in [71] with approximation ratio of $3e$ (≈ 8.16). When all coflows have release dates equal to zero, our deterministic algorithm has approximation ratio of 4 while the best prior known result is 8 [70, 71] for deterministic and $2e$ (≈ 5.436) [71] for randomized algorithms [1]. We would like to mention that, although the randomized algorithm in [68], has a better approximation ratio of $2 + \epsilon$ compared to this work, the algorithm in this chapter still provides the best approximation ratio when the rate allocation is restricted to data units (i.e., matching constraint). See Section 3.2 for more details on the model.

• **Empirical Evaluations.** We evaluate the performance of our algorithm, compared to the prior

[1]The paper [76] proposes an algorithm with the same approximation guarantee, however, it uses a different linear programming, and our scheduling policy is much simpler than the policy they proposed. Moreover, we also study the performance of our algorithm through extensive simulations with synthetic and real traffic traces and compare its performance with other coflow scheduling algorithms.

Table 3.1: Performance guarantees (Approximation ratios)

Case	Best known		This work
	deterministic	randomized	deterministic
Without release dates	8 [70, 71]	$2e$ [71]	4
With release dates	12 [70, 71]	$3e$ [71]	5

approaches, using both syntectic traffic as well as real traffic based on a Hive/MapReduce trace from a large production cluster at Facebook. Both synthetic and empirical evaluations show that our deterministic algorithm indeed outperforms the prior approaches. For instance, for the Facebook trace with general release dates, our algorithm outperforms Varys [8], the algorithm proposed in [69], and the algorithm proposed in [71] by 24%, 40%, and 19%, respectively. Finally, we compare the fairness of various algorithms and propose couple of ideas to improve the fairness. The result presented in this chapter is based on papers [71, 77, 78].

3.2 System Model and Problem Formulation

Datecenter Network: Similar to [8, 69], we abstract out the data center network as one giant $N \times N$ non-blocking switch, with N input links connected to N source servers and N output links connected to N destination servers. Thus the network can be viewed as a bipartite graph with source nodes denoted by set $\mathcal{I}$ on one side and destination nodes denoted by set $\mathcal{J}$ on the other side (therefore, $\mathcal{I} \cap \mathcal{J} = \emptyset$.). Moreover, there are capacity constraints on the input and output links. For simplicity, we assume all links have equal capacity (as in [69]); nevertheless, our method can be easily extended to the general case where the links have unequal capacities. Without loss of generality, we assume that all the link capacities are normalized to one. *Scheduling Constraints:* We allow a general class of scheduling algorithms where the rate allocation can be performed continuously over time, i.e., for each flow, fractional data units can be transferred from its input link to its corresponding output link over time as long as link capacity constraints are respected. In the special case that the rate allocation is restricted to data units (packets), each source node can send at most one packet in every time unit (time slot) and each destination node can receive

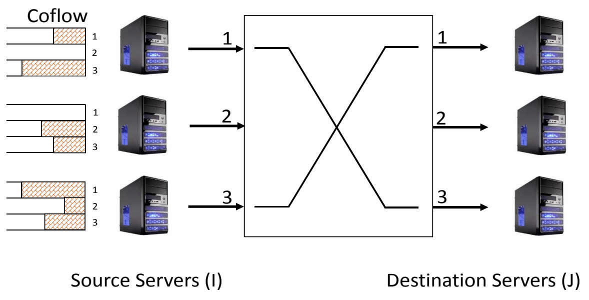


Figure 3.1: A coflow in a 3×3 switch architecture.

at most one packet in every time slot, and the feasible schedule has to form a matching of the switch's bipartite graph (matching constraint). In this case, our model reduces to the model in [69] and, as it is shown later, our proposed algorithm will respect the matching constraints, therefore, it is compatible with both models.

Coflow: A coflow is a collection of flows whose completion time is determined by the completion time of the latest flow in the collection. The coflow k can be denoted as an $N \times N$ demand matrix $D^{(k)}$. Every flow is a triple (i, j, k), where $i \in \mathcal{I}$ is its source node, $j \in \mathcal{J}$ is its destination node, and k is the coflow to which it belongs. The size of flow (i, j, k) is denoted by d_{ij}^{k}, which is the (i, j)-th element of the matrix $D^{(k)}$. For simplicity, we assume that all flows within a coflow arrive to the system at the same time (as in [69]); however, our results still hold for the case that flows of a coflow are released at different times (which could indeed happen in practice [10]). A 3×3 switch architecture is shown in Figure 3.1 as an example, where a coflow is illustrated by means of input queues, e.g., the file in the j-th queue at the source link i indicates that the coflow has a flow from source server i to destination server j. For instance, in Figure 3.1, the illustrated coflow has 7 flows in total, while two of its flows have source server 1, one goes to destination server 1 and the other to destination server 3.

Total Weighted Coflow Complettion Time: We consider the coflow scheduling problem with release dates. There is a set of K coflows denoted by $\mathcal{K}$. Coflow $k \in \mathcal{K}$ is released (arrives) at time r_k which means it can only be scheduled after time r_k. We use f_k to denote the finishing (completion) time of coflow k, which, by definition of coflow, is the time when all its flows have finished processing. In other words, for every coflow $k \in \mathcal{K}$,

$$f_k = \max_{i \in \mathcal{I}, j \in \mathcal{J}} f_{ij}^k, \tag{3.1}$$

where f_{ij}^k is the completion time of flow (i, j, k).

For given positive weights w_k, $k \in \mathcal{K}$, the goal is to minimize the weighted sum of coflow completion times: $\sum_{k=1}^K w_k f_k$. The weights can capture different priority for different coflows. In the special case that all the weights are equal, the problem is equivalent to minimizing the average coflow completion time.

Define

$$T = \max_{k \in \mathcal{K}} r_k + \sum_{k \in \mathcal{K}} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} d_{ij}^k. \tag{3.2}$$

Note that T is clearly an upper bound on the minimum time required for processing of all the coflows. We denote by $x_{ij}^k(t)$ the transmission rate assigned to flow (i, j, k) at time $t \in [0, T]$. Then the optimal rate control must solve the following optimal control problem

$$\text{minimize} \sum_{k=1}^{K} w_k f_k \tag{3.3a}$$

$$\text{subject to: } f_k \geq f_{ij}^k, \quad i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K} \tag{3.3b}$$

$$d_{ij}^k = \int_0^{f_{ij}^k} x_{ij}^k(t) dt, \quad i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K} \tag{3.3c}$$

$$\sum_j \sum_k x_{ij}^k(t) \leq 1, \quad i \in \mathcal{I}, t \in [0, T] \tag{3.3d}$$

$$\sum_i \sum_k x_{ij}^k(t) \leq 1, \quad j \in \mathcal{J}, t \in [0, T] \tag{3.3e}$$

$$x_{ij}^k(t) = 0, \quad \forall t < r_k, \; i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K} \tag{3.3f}$$

$$x_{ij}^k(t) \geq 0, \quad i \in \mathcal{I}, j \in \mathcal{J}, k \in \mathcal{K}, t \in [0, T] \tag{3.3g}$$

In the above, the constraint (3.3b) indicates that each coflow k is completed when all its flows have been completed. Note that since the optimization (3.3) is a minimization problem, a coflow completion time is equal to the completion time of its latest flow, in agreement with (3.1). The constraint (3.3c) ensures that the demand (file size) of every flow, d_{ij}^k, is transmitted by its completion time, f_{ij}^k. Constraints (3.3d) and (3.3e) state the capacity constraints on source links and destination links, respectively. The fact that a flow cannot be transmitted before its release date (which is equal to release date of its corresponding coflow) is captured by the constraint (3.3f). Finally, the constraint (3.3g) simply states that the rates are non-negative.

Remark 2. *An alternative formulation of (3.3) could be minimizing the weighted sum of delays, where delay of coflow k is defined as $f_k - r_k$. The two minimizations are equivalent as only the objectives differ in a constant term $\sum_k r_k w_k$, however in terms of approximation results they could be very different. In the case of zero release dates, the two formulations are trivially the same, and our algorithms yield the same approximation results for both formulations. However, for the general release dates, there is no constant ratio approximation algorithm for minimizing the weighted sum of delays. This can be shown through its connection to the single machine scheduling for which finding a constant approximation algorithm for the delay-based formulation is NP-complete [79].*

3.3 Motivations and Challenges

The coflows can be widely different in terms of the number of parallel flows, the size of individual flows, the groups of servers involved, etc. Heuristics from traditional flow/task scheduling, such as shortest- or smallest-first policies [80, 81], do not have a clear equivalence in coflow scheduling. One can define a shortest or smallest-first policy based on the number of parallel flows in a coflow, or the aggregate flow sizes in a coflow, however these policies perform poorly [8], as they do not completely take all the characteristics of coflows into consideration.

Recall that the completion time of a coflow is dominated by its slowest flow (as described by (3.1) or (3.3b)). Hence, it makes sense to slow down all the flows in a coflow to match the completion time of the flow that will take the longest to finish. The unused capacity then can be used to allow other coexisting coflows to make progress and the total (or average) coflow completion time decreases. Varys [8] is the first heuristic that effectively implements this intuition by combining Smallest-Effective-Bottleneck-First and Minimum-Allocation-for-Desired-Duration policies. Before describing Varys, we present a few definitions that are used in the rest of this chapter.

Definition 2 (Aggregate Size and Effective Size of a Coflow). *Let*

$$d_i^k = \sum_{j \in \mathcal{J}} d_{ij}^k; \;\; d_j^k = \sum_{i \in \mathcal{I}} d_{ij}^k, \tag{3.4}$$

be respectively the aggregate flow size that coflow k needs to send from source node i and receive at destination node j. The effective size of coflow k is defined as

$$W(k) = \max\{\max_{i \in \mathcal{I}} d_i^k, \max_{j \in \mathcal{J}} d_j^k\}. \tag{3.5}$$

Thus $W(k)$ is the maximum amount of data that needs to be sent or received by a node for coflow k. Note that, due to normalized capacity constraints on links, *when coflow k is released, we need at least $W(k)$ amount of time to process all its flows.*

Overview of Varys. Varys [8] orders coflows in a list based on their effective size in an increasing order. Transmission rates of individual flows of the first coflow in the list are set such that all its flows complete at the same time. The remaining capacity of links are updated and iteratively distributed among other coflows in the list in a similar fashion. Formally, the completion time of coflow k, $k = 1, \ldots, K$, is calculated as follows

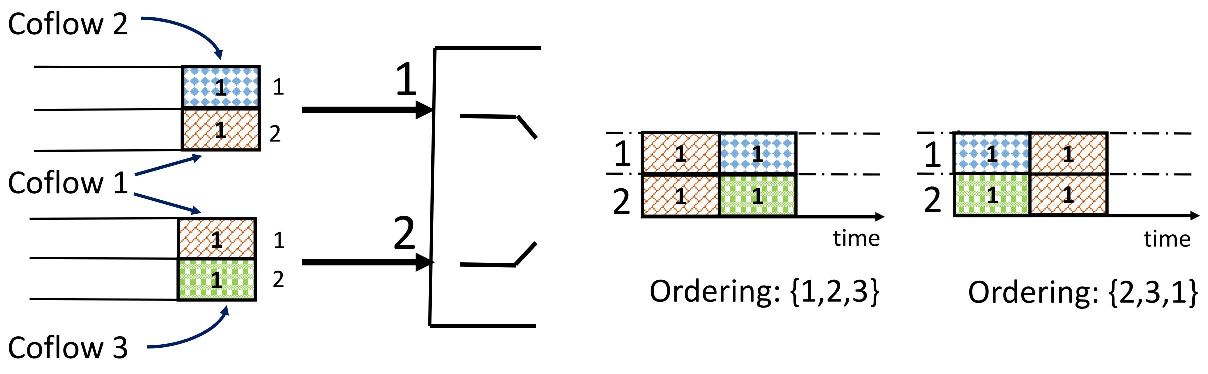
$$\Gamma^k = \max\{\max_{i \in \mathcal{I}} \frac{d_i^k}{Rem(i)}, \max_{j \in \mathcal{J}} \frac{d_j^k}{Rem(j)}\},$$

where $Rem(i)$ (similarly, $Rem(j)$) is the remaining capacity of input link i (output link j) after transmission rates of all coflows $k' < k$ are set. Then for flow (i, j, k), Varys assigns transmission rate $x_{ij}^k = d_{ij}^k / \Gamma^k$. In case that there is still idle capacity, for each input link $i \in \mathcal{I}$, the remaining capacity is allocated to the flows of coflows subject to capacity constraints in corresponding output links. Once the first coflow completes, all the flow sizes and the scheduling list are updated and the iterative procedure is repeated to complete the second coflow and distribute the unused capacity. The procedure is stopped when all the coflows are processed.

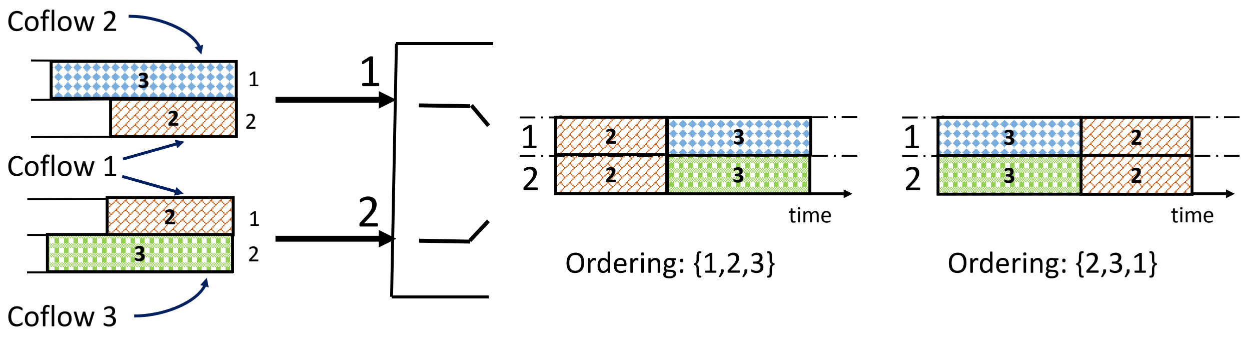
While Varys performs better than traditional flow scheduling algorithms, it could still be inefficient. The main reason is that *Varys is oblivious to the dependency among coflows that share a (source or destination) node*. To further expose this issue, we present a simple example.

Example 1 (Inefficiency of Varys)**.** *Consider the* 2×2 *switch network illustrated in Figure 3.2 where there are* 3 *coflows in the system. In Figure 3.2a, the effective coflow sizes are* $W(1) = W(2) = W(3) = 1$*, therefore, Varys cannot differentiate among coflows. Scheduling coflows in the order* $\{1, 2, 3\}$ *or* $\{2, 3, 1\}$ *are both possible under Varys but they result in different total completion times,* $1 + 2 + 2 = 5$ *and* $1 + 1 + 2 = 4$*, respectively (assuming the weights are all one for all the coflows). Next, consider a slight modification of flow sizes, as shown in Figure 3.2b. In this example* $W(1) = 2$ *and* $W(2) = W(3) = 3$*. Based on Varys algorithm, coflow* 1 *is scheduled first during time interval* $(0, 2]$ *at rate* 1*. When coflow* 1 *completes, coflows* 2 *and* 3 *are scheduled in time interval* $(2, 5]$*; hence, the total completion time will be* $2 + 5 + 5 = 12$*. However, if we schedule coflows* 2 *and* 3 *first, the total completion times will reduce to* $3 + 3 + 5 = 11$*. Note that in both examples, coflow* 1 *completely blocks coflows* 2 *and* 3*, which is not captured by Varys. In fact, the negative impact of ignoring configuration of coflows and their shared nodes is much more profound in large networks with a large number of coflows (see simulations in Section 3.8).*

Overview of LP-based algorithms. The papers [69] and [71] use Linear Programs (LPs) (based on interval-indexed variables or ordering variables) that capture more information about coflows and provide a better ordering of coflows for scheduling compared to Varys [8]. At the high level, the technical approach in these papers is based on partitioning jobs (coflows) into polynomial number of groups based on solution to a polynomial-sized relaxed linear program, and minimizing the completion time of each group by treating the group as a single coflow. Grouping can have a significant impact on decreasing the completion time of coflows. For instance, in view of examples in Figure 3.2, grouping coflows 2 and 3, and scheduling them first, decreases the total completion



(a) All coflows have equal effective size. Both orderings are possible under Varys, with the total completion time of $1 + 2 + 2 = 5$ and $1 + 1 + 2 = 4$, for the left and right ordering respectively.



(b) Varys schedules coflow 1 first, according to the ordering $\{1, 2, 3\}$, which gives a total completion time of $2 + 5 + 5 = 12$. The optimal schedule is the ordering $\{2, 3, 1\}$ with a total completion time of $3 + 3 + 5 = 11$.

Figure 3.2: Inefficiency of Varys in a 2×2 switch network with 3 coflows.

time as explained.

NP-hardness and connection to the concurrent open shop problem. The concurrent open shop problem [75] can be essentially viewed as a special case of the coflow scheduling problem *when demand matrices are diagonal* (in the jargon of concurrent open shop problem, the coflows are jobs, the flows in each coflow are tasks for that job, and the destination nodes are machines with unit capacities). It is known that it is NP-complete to approximate the concurrent open shop problem, when jobs are released at time zero, within a factor better than $2 - \epsilon$ for any $\epsilon > 0$ [82]. Although the model we consider for coflow scheduling is different from the model used in [69], similar reduction as proposed in [69] can be leveraged to show NP-completness of the coflow scheduling problem. More precisely, every instance of the concurrent open shop problem can be reduced to an instance of coflow scheduling problem (see Section 3.9 for the details), hence it is

NP-complete to approximate the coflow scheduling problem (without release dates) within $2 - \epsilon$, for any $\epsilon > 0$. There are 2-approximation algorithms for the concurrent open shop (e.g., [75]), however, these algorithms cannot be ported to the coflow scheduling problem due to the coupling of source and destination link capacity constraints in the coflow scheduling problem (see Section 3.9 for a counter example that shows the 2-approximation algorithm from concurrent open shop cannot be ported to the coflow scheduling problem).

Next, we describe our coflow scheduling algorithm. The algorithm is based on a linear program formulation for sorting the coflows followed by a simple list scheduling policy

3.4 Linear Programing (LP) Relaxation

In this section, we use *linear ordering variables* (see, e.g., [83, 74, 84, 75]) to present a relaxed integer program of the original scheduling problem (3.3). We then relax these variables to obtain a linear program (LP). In the next section, we use the optimal solution to this LP as a subroutine in our scheduling algorithm.

Ordering variables. For each pair of coflows, if both coflows have *some* flows incident at some node (either originated from or destined at that node), we define a binary variable which indicates which coflow finishes all its flows before the other coflow does so in the schedule. Formally, for any two coflows k, k' with aggregate flow sizes $d_m^k \neq 0$ and $d_m^{k'} \neq 0$ on some node $m \in \mathcal{I} \cup \mathcal{J}$ (recall definition (3.4)), we introduce a binary variable $\delta_{kk'} \in \{0, 1\}$ such that $\delta_{kk'} = 1$ if coflow k finishes all its flows before coflow k' finishes all its flows, and it is 0 otherwise. If both coflows finish their flows at the same time (which is possible in the case of continuous-time rate control), we set either one of $\delta_{kk'}$ or $\delta_{k'k}$ to 1 and the other one to 0, arbitrarily.

Relaxed Integer Program (IP). We formulate the following Integer Program (IP):

$$\textbf{(IP)} \ \min \ \sum_{k=1}^{K} w_k f_k \tag{3.6a}$$

$$f_k \geq d_i^k + \sum_{k' \in \mathcal{K}} d_i^{k'} \delta_{k'k} \ \ i \in \mathcal{I}, k \in \mathcal{K} \tag{3.6b}$$

$$f_k \geq d_j^k + \sum_{k' \in \mathcal{K}} d_j^{k'} \delta_{k'k} \ \ j \in \mathcal{J}, k \in \mathcal{K} \tag{3.6c}$$

$$f_k \geq W(k) + r_k \ \ k \in \mathcal{K} \tag{3.6d}$$

$$\delta_{kk'} + \delta_{k'k} = 1 \ \ k, k' \in \mathcal{K} \tag{3.6e}$$

$$\delta_{kk'} \in \{0, 1\} \ \ k, k' \in \mathcal{K} \tag{3.6f}$$

In the above, to simplify the formulation, we have defined $\delta_{kk'}$, for all pairs of coflows, by defining $d_m^k = 0$ if coflow k has no flow originated from or destined to node m.

The constraint (3.6b) (similarly (3.6c)) follows from the definition of ordering variables and the fact that flows incident to a source node i (a destination node j) are processed by a single link of unit capacity. To better see this, note that the total amount of traffic can be sent in the time period $(0, f_k]$ over the i-th link is at most f_k. This traffic is given by the right-hand-side of (3.6b) (similarly (3.6c)) which basically sums the aggregate size of coflows incident to node i that finish their flows before coflow k finishes its corresponding flows, plus the aggregate size of coflow k at node i itself, d_i^k. This implies constraint (3.6b) and (3.6c). The fact that each coflow cannot be completed before its release date plus its effective size is captured by constraint (3.6d). The next constraint (3.6e) indicates that for each two incident coflows, one precedes the other.

Note that this optimization problem is a relaxed integer program for the problem (3.3), since the set of constraints are not capturing all the requirements we need to meet for a feasible schedule. For example, we cannot start scheduling flows of a coflow when it is not released yet, while constraint (3.6d) does not necessarily avoid this, thus leading to a smaller value of finishing time compared to the optimal solution to (3.3). Further, release dates and scheduling constraints in optimization (3.3) might cause idle times in flow transmission of a node, therefore yielding a larger

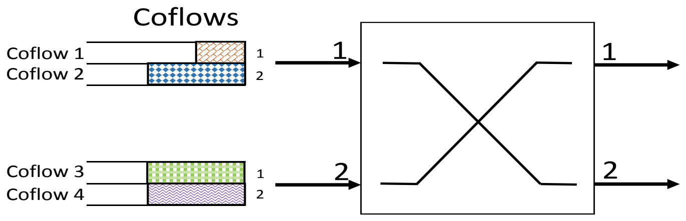


Figure 3.3: 4 coflows in a 2×2 switch architecture, flow $(1, 1)$ is released at time 0, and all the others are released at time 1.

value of finishing time for a coflow than what is restricted by (3.6b), (3.6c), (3.6d). To further illustrate this issue, we present a simple example.

Example 2. *Consider a 2×2 switch network as in Figure 3.3. Assume there are 4 coflows, each has one flow. Flow $(1, 1, 1)$ is released at time 0 with size 1, and the other three flows are released at time 1 with size 2. It is easy to check that the following values for the ordering variables and flow completion times satisfy all the constraints (3.6b)–(3.6f). For brevity, we only report the ordering variables for coflows that actually share a node. For example, it is redundant to consider ordering variables corresponding to coflow 1 and coflow 4 as they are not incident at any (source/destination) node and any value for their associated pairwise ordering variables does not have any impact on the optimal value for IP (3.6). Below, the ordering variables and coflow completion times are presented, and all the ordering variables which are not specified can be taken as zero.*

$$\delta_{12} = 1, \quad \delta_{34} = 1,$$

$$\delta_{13} = 1, \quad \delta_{24} = 1,$$

$$f_1 = 1, \quad f_2 = 3,$$

$$f_3 = 3, \quad f_4 = 4.$$

While these values satisfy (3.6b)–(3.6f), this is not a valid schedule since it requires transmission of flow $(2, 2, 4)$ starting at time 0, while it is not released yet. To see this, note that $f_1 = 1$, so to finish processing of coflow 1 or equivalently flow $(1, 1, 1)$ by time 1, we need to start its transmission at maximum rate at time 0. Then, due to the capacity constraints, the first time flows $(1, 2, 2)$ and $(2, 1, 3)$ can start transmission is at time 1, when flow $(1, 1, 1)$ has been completed. Since we require *to complete both of these flows at time* 3*, they need to be transmitted at maximum rate in the time interval* $(1, 3]$*. Therefore, the only way to finish flow* $(2, 2, 4)$ *at time* 4 *is to send one unit of its data in time interval* $(0, 1]$ *and its remaining unit of data in time interval* $(3, 4]$*, but this flow has not been released before time* 1*. So the proposed IP does not address all the scheduling constraints.*

Relaxed Linear Program (LP). In the linear program relaxation, we allow the ordering variables to be fractional. Specifically, we replace the constraints (3.6f) with the constraints (3.7b) below. We refer to the obtained linear problem by (LP).

$$(\mathbf{LP}) \ \min \ \sum_{k=1}^{K} w_k f_k \tag{3.7a}$$

$$\text{subject to: (3.6b) – (3.6e)},$$

$$\delta_{kk'} \in [0, 1] \ \ k, k' \in \mathcal{K} \tag{3.7b}$$

We use $\tilde{f}_k$ to denote the optimal solution to this LP for the completion time of coflow k. Also we use $\widetilde{\text{OPT}} = \sum_k w_k \tilde{f}_k$ to denote the corresponding objective value. Similarly we use $f_k^\star$ to denote the optimal completion time of coflow k in the original coflow scheduling problem (3.3), and use $\text{OPT} = \sum_k w_k f_k^\star$ to denote its optimal objective value. The following lemma establishes a relation between $\widetilde{\text{OPT}}$ and OPT.

Lemma 6. *The optimal value of the LP,* $\widetilde{OPT}$*, is a lower bound on the optimal total weighted completion time OPT of coflow scheduling problem.*

Proof. Consider an optimal solution to the optimization problem (3.3). We set ordering variables so as $\delta_{kk'} = 1$ if coflow k precedes coflow k' in this solution, and $\delta_{kk'} = 0$, otherwise. If both coflows finish their corresponding flows at the same time, we set either one to 1 and the other one to 0. We note that this set of ordering variables and coflow completion times satisfies constraints (3.6b) and (3.6c) (by taking integral from both side of constraint (3.3d) and (3.3e) from time 0 to f_k) and also constraint (3.6d) (by combining constraints (3.3c) and (3.3f)). Furthermore, the rest of (LP) constraints are satisfied by the construction of ordering variables. Therefore, optimal solution of problem (3.3) can be converted to a feasible solution to (LP). Hence, the optimal value of LP, $\widetilde{\text{OPT}}$, is at most equal to OPT. □

3.5 Coflow Scheduling Algorithm

In this section, we describe our polynomial-time coflow scheduling algorithm and state the main results about its performance guarantees.

The scheduling algorithm is presented in Algorithm 3. It has three main steps:

1. solve the relaxed LP (3.7),

2. use the solution of the relaxed LP to order flows of coflows,

3. apply a simple list scheduling algorithm based on the ordering.

The relaxed LP (3.7) has $O(K^2)$ variables and $O(K^2 + KN))$ constraints and can be solved efficiently in polynomial time, e.g. using interior point method [85] (see Section 3.8.5 for more details about the complexity).

Then, the algorithm orders the coflows based on values of $\tilde{f}_k$ (optimal solution to LP) in non-decreasing order. More precisely, we re-index coflows such that,

$$\tilde{f}_1 \leq \tilde{f}_2 \leq \ldots \leq \tilde{f}_K. \tag{3.8}$$

Ties are broken arbitrarily. We emphasize that we do not need to round the values of the ordering variables in LP to obtain the ordering of coflows, instead we use the values of $\tilde{f}_k$ (optimal solution to LP) which do not need to be integer.

At any time, the algorithm maintains a list for the flows in the system such that for every two flows (i, j, k) and (i', j', k') with $k < k'$ (based on ordering (3.8)), flow (i, j, k) is placed before flow (i', j', k') in the list. Flows of the same coflow are listed in an arbitrary order. The algorithm scans the list starting from the first flow and schedules a flow if both its corresponding source and destination links are idle at that time. Upon completion of a flow or arrival of a coflow, the algorithm preempts the schedule, updates the list, and starts scheduling the flows based on the updated list.

Algorithm 3 Deterministic Coflow Scheduling Algorithm

```
Suppose coflows $\left[d_{ij}^k,\ 1 \le i,j \le N\right]$, $k \in \mathcal{K}$, with release dates $r_k$, $k \in \mathcal{K}$, and weights $w_k$,
$k \in \mathcal{K}$, are given.
 1: Solve the linear program (LP) and denote its optimal solution by $\{\tilde{f}_k; k \in \mathcal{K}\}$.
 2: Order and re-index the coflows such that:
        $$\tilde{f}_1 \le \tilde{f}_2 \le \ldots \le \tilde{f}_K, \qquad (3.9)$$
    where ties are broken arbitrarily.
 3: Wait until the first coflow is released.
 4: while There is some incomplete flow, do
 5:     List the released and incomplete flows respecting the ordering in (3.9). Let $L$ be the
        total number of flows in the list.
 6:     for $l = 1$ to $L$ do
 7:         Denote the $l$-th flow in the list by $(i_l, j_l, k_l)$,
 8:         if Both the links $i_l$ and $j_l$ are idle, then
 9:             Schedule flow $(i_l, j_l, k_l)$.
10:         end if
11:     end for
12:     while No new flow is completed or released do
13:         Transmit the flows that get scheduled in line 9 at maximum rate 1.
14:     end while
15: end while
```

The main result regarding the performance of Algorithm 3 is stated in Theorem 2.

Theorem 2. *Algorithm 3 is a polynomial-time* 5*-approximation algorithm for the problem of minimizing total weighted completion time of coflows with release dates.*

When all coflows are released at time 0, we can improve the algorithm's performance ratio.

Corollary 1. *If all coflows are released at time* 0*, then Algorithm 3 is a* 4*-approximation algorithm.*

3.6 Proof Sketch of Main Results

In this section, we present the sketch of proofs of the main results for our polynomial-time coflow scheduling algorithm. Before proceeding with the proofs, we make the following definitions.

Definition 3 (Aggregate Size and Effective Size of a List of Coflows)**.** *For a list of K coflows and for a node $s \in \mathcal{I} \cup \mathcal{J}$, we define $W(1, \cdots, k; s)$ to be the amount of data needs to be sent or received by node s in the network considering only the first k coflows. We also denote by $W(1, \cdots, k)$ the effective size of the aggregate coflow constructed by the first k coflows, $k \leq K$. Specifically,*

$$W(1, \cdots, k; s) = \sum_{l=1}^{k} d_s^l \tag{3.10}$$

$$W(1, \cdots, k) = \max_{s \in \mathcal{I} \cup \mathcal{J}} W(1, \cdots, k; s) \tag{3.11}$$

3.6.1 Bounded Completion Time for The Collection of Coflows

Consider the list of coflows according to the ordering in (3.8) and define $W(1, \cdots, k)$ based on Definition 3. The following lemma demonstrates a relationship between completion time of coflow k obtained from (LP) and $W(1, \cdots, k)$ which is used later in the proofs.

Lemma 7. $\tilde{f}_k \geq \frac{W(1,\cdots,k)}{2}$.

Proof. The proof uses similar ideas as in Gandhi, et al. [84] and Kim [73]. Using constraint (3.6b), for any source node $i \in \mathcal{I}$, we have

$$d_i^l \tilde{f}_l \geq (d_i^l)^2 + \sum_{l' \in \mathcal{K}} d_i^l d_i^{l'} \delta_{l'l} \tag{3.12}$$

which implies that,

$$\begin{aligned}\sum_{l=1}^{k} d_i^l \tilde{f}_l \geq& \sum_{l=1}^{k} (d_i^l)^2 + \sum_{l=1}^{k} \sum_{l'=1}^{k} d_i^{l'} d_i^l \delta_{l'l} \\ =& \frac{1}{2}\Bigg(2 \times \sum_{l=1}^{k} (d_i^l)^2 \\ &+ \sum_{l=1}^{k} \sum_{l'=1}^{k} \left(d_i^{l'} d_i^l \delta_{l'l} + d_i^{l'} d_i^l \delta_{ll'}\right)\Bigg)\end{aligned} \tag{3.13}$$

We simplify the right-hand side of (3.13), using constraint (3.6e), combined with the following equality

$$\sum_{l=1}^{k}(d_i^l)^2 + \sum_{l=1}^{k}\sum_{l'=1}^{k} d_i^{l'} d_i^l = (\sum_{l=1}^{k} d_i^l)^2, \tag{3.14}$$

and conclude that

$$\begin{aligned}\sum_{l=1}^{k} d_i^l \tilde{f}_l \geq & \frac{1}{2}\sum_{l=1}^{k}(d_i^l)^2 + \frac{1}{2}(\sum_{l=1}^{k} d_i^l)^2 \\ \geq & \frac{1}{2}(\sum_{l=1}^{k} d_i^l)^2 = \frac{1}{2}(W(1,\cdots,k;i))^2\end{aligned} \tag{3.15}$$

Where the last equality follows from Definition 3.10. Similar argument results in the following inequality for any destination node $j \in \mathcal{J}$, i.e.,

$$\sum_{l=1}^{k} d_j^l \tilde{f}_l \geq \frac{1}{2}(W(1,\cdots,k;j))^2.$$

Now consider the node $s^\star$ which has the maximum load induced by the first k coflows, namely, $W(1,\cdots,k) = W(1,\cdots,k;s^\star)$.

$$\begin{aligned}\tilde{f}_k W(1,\cdots,k;s^\star) &= \tilde{f}_k \sum_{l=1}^{k} d_{s^\star}^l \\ &\geq \sum_{l=1}^{k} d_{s^\star}^l \tilde{f}_l \\ &\geq \frac{1}{2}(W(1,\cdots,k;s^\star))^2\end{aligned} \tag{3.16}$$

This implies that,

$$\tilde{f}_k \geq \frac{1}{2}W(1,\cdots,k;s^\star) = \frac{1}{2}W(1,\cdots,k). \tag{3.17}$$

This completes the proof. □

Note that $W(1,\cdots,k)$ is a lower bound on the time that it takes for all the first k coflows to be completed (as a result of the capacity constraints in the optimization (3.3)). Hence, Lemma 7 states that by allowing ordering variables to be fractional, completion time of coflow k obtained from (LP) is still lower bounded by half of $W(1, \cdots, k)$.

3.6.2 Proof of Theorem 2 and Corollary 1

Proof of Theorem 2. We use $\{\hat{f}_k\}_{k=1}^K$ to denote the actual coflow completion times under our deterministic algorithm. Suppose flow (i, j, k) is the last flow of coflow k that is completed. In general, Algorithm 3 may preempt a flow several times during its execution. For now, suppose flow (i, j, k) is not preempted and use t_k to denote the time when its transmission is started (the arguments can be easily extended to the preemption case as we show at the end of the proof). Therefore

$$\hat{f}_k = \hat{f}_{ij}^k = t_k + d_{ij}^k \tag{3.18}$$

From the algorithm description, t_k is the first time that both links i and j are idle and there are no higher priority flows to be scheduled (i.e., there is no flow (i, j, k') from i to j with $k' < k$ in the list). By definition of $W(1, \cdots, k; s)$, node s, $s \in \{i, j\}$, has $W(1, \cdots, k; s) - d_{ij}^k$ data units to send or receive by time t_k. Since the capacity of all links are normalized to 1, it should hold that

$$\begin{aligned} t_k &\le r_k + W(1, \cdots, k; i) - d_{ij}^k + W(1, \cdots, k; j) - d_{ij}^k \\ &\le r_k + 2W(1, \cdots, k) - 2d_{ij}, \end{aligned}$$

where the last inequality is by Definition 3.11. Combining this inequality with equality (3.18) yields the following bound on $\hat{f}_k$.

$$\hat{f}_k \le r_k + 2W(1, \cdots, k)$$

Using Lemma 7 and constraint (3.6d), we can conclude that

$$\hat{f}_k \le 5\tilde{f}_k,$$

which implies that

$$\sum_{k=1}^{K} w_k \hat{f}_k \leq 5 \sum_{k=1}^{K} w_k \tilde{f}_k.$$

This shows an approximation ratio of 5 for Algorithm 3 using Lemma 6. Finally, if flow (i, j, k) is preempted, the above argument can still be used by letting t_k to be the starting time of its last piece and d_{ij}^k to be the remaining size of its last piece at time t_k. This completes the proof. □

Proof of Corollary 1. When all coflows are released at time 0, $t_k \leq W(1, \cdots, k) - d_{ij}^k + W(1, \cdots, k) - d_{ij}^k$. The rest of the argument is similar to the proof of Theorem 2. Therefore, the algorithm has approximation ratio of 4 when all coflows are release at time 0. □

3.7 Extension to Online Algorithm

Similar to previous work [69, 70], Algorithm 3 is an offline algorithm, and requires the complete knowledge of the flow sizes and release dates. While this knowledge can be learned in long running services, developing online algorithms that deal with the dynamic nature and unavailability of this information is of practical importance. One natural extension of our algorithm to an online setting, assuming that the coflow information revealed at its release date, is as follows: Upon each coflow arrival, we re-order the coflows by re-solving the (LP) using the remaining coflow sizes and the newly arrived coflow, and update the list. Given the updated list, the scheduling is done as in Algorithm 3. To reduce complexity of the online algorithm, we may re-solve the LP once in every T seconds, for some T that can be tuned, and update the list accordingly. We leave theoretical and experimental study of this online algorithm as a future work.

3.8 Empirical Evaluations

In this section, we present our simulation results and evaluate the performance of our algorithm for both cases of with and without release dates, under both synthetic and real traffic traces. We also simulate the deterministic algorithms proposed in [71, 69] and Varys [8] and compare their performance with the performance of our algorithm. Finally, we comment on the fairness issues of the algorithm.

3.8.1 Workload

We evaluate algorithms under both synthetic and real traffic traces.

Synthetic traffic: To generate synthetic traces we slightly modify the model used in [86]. We consider the problem of size $K = 160$ coflows in a switch network with $N = 16$ input and output links. We denote by M the number of non-zero flows in each coflow. We consider two cases:

- Dense instance: For each coflow, M is chosen uniformly from the set $\{N, N + 1, ..., N^2\}$. Therefore, coflows have $O(N^2)$ non-zero flows on average.
- Combined instance: Each coflow is sparse or dense with probability $1/2$. For each sparse coflow, M is chosen uniformly from the set $\{1, 2, ..., N\}$, and for each dense coflow M is chosen uniformly from the set $\{N, N + 1, ..., N^2\}$.

Given the number M of flows in each coflow, M pairs of input and output links are chosen randomly. For each pair that is selected, an integer flow size (processing requirement) d_{ij} is randomly selected from the uniform distribution on $\{1, 2, ..., 100\}$. For the case of scheduling with release dates, we generate the coflow inter-arrival times uniformly from $[1, 100]$. We generate 100 instances for each case and report the average algorithms' performance.

Real traffic: This workload was also used in [8, 69, 71]. The workload is based on a Hive/MapReduce trace at Facebook that was collected from a 3000-machine cluster with 150 racks. In this trace, the following information is provided for each coflow: arrival time of the coflow in millisecond, locations of mappers (rack number to which they belong), locations of reducers (rack number to which they belong), and the amount of shuffle data in Megabytes for each reducer. We assume that shuffle data of each reducer in a coflow is evenly generated from all mappers specified for that coflow. The data trace consists of 526 coflows in total from very sparse coflows (the most sparse coflow has only 1 flow) to very dense coflows (the most dense coflow has 21170 flows.). Similar to [69], we filter the coflows based on the number of their non-zero flows, M. Apart from considering all coflows ($M \geq 1$), we consider three coflow collections filtered by the conditions $M \geq 10$, $M \geq 30$, and $M \geq 50$. In other words, we use the following 4 collections:

- All coflows,
- Coflows with $M \geq 10$,
- Coflows with $M \geq 30$,
- Coflows with $M \geq 50$.

Furthermore, the original cluster had a 10:1 core-to-rack oversubscription ratio with a total bisection bandwidth of 300 Gbps. Hence, each link has a capacity of 128 MBps. To obtain the same traffic intensity offered to our network (without oversubscription), for the case of scheduling coflows with release dates, we need to scale down the arrival times of coflows by 10. For the case of without release dates, we assume that all coflows arrive at time 0.

3.8.2 Algorithms

We simulate four algorithms: the algorithm proposed in this chapter, Varys [8], the deterministic algorithm in [69], and the deterministic algorithm in [71]. We briefly overview these algorithms and also elaborate on the backfilling strategy that has been combined with the deterministic algorithms in [69, 71] to avoid under utilization of network resources.

1. Varys [8]: Scheduling and rate assignments under Varys were explained in detail in Section 3.3. There is a parameter δ in the original design of Varys that controls the tradeoff between fairness and completion time. Since we focus on minimizing the total completion time of coflows, we set δ to 0 which yields the best performance of Varys. In this case, upon arrival or completion of a coflow, the coflow ordering is updated and the rate assignment is done iteratively as described in Section 3.3.

2. Interval-Indexed-Grouping (LP-II-GB) [69]: The algorithm requires discrete time (i.e., time slots) and is based on an interval-indexed formulation of a polynomial-time linear program

(LP) as follows. The time is divided into geometrically increasing intervals. The binary decision variables x_{lk} are introduced which indicate whether coflow k is scheduled to complete within the l-th interval $(t_l, t_{l+1}]$. Using these binary variables, a lower bound on the objective function is formulated subject to link capacity constraints and the release date constraints. The binary variables are then relaxed leading to an LP whose solution is used for ordering coflows. More precisely, the relaxed completion time of coflow k is defined as $f_k = \sum_l t_l x_{lk}$, where t_l is the left point of the l-th interval and $x_{lk} \in [0, 1]$ is the relaxed decision variable. Based on the optimal solution to this LP, coflows are listed in an increasing order of their relaxed completion time. For each coflow k in the list, $k = 1, ..., K$, we compute effective size of the cumulated first k coflows in the list, $W(1, \cdots, k)$. All coflows that fall within the same time interval according to value of $W(1, \cdots, k)$ are grouped together and treated as a single coflow and scheduled so as to minimize its completion time. Scheduling of coflows within a group makes use of the Birkhoff-von Neumann decomposition. If two data units from coflows k and k' within the same group use the same pair of input and output, and k is ordered before k', then we always process the data unit from coflow k first. For backfilling, when we use a schedule that matches input i to output j, if there is no more service requirement on the pair of input i and output j for some coflow in the current partition, we backfill in order from the flows on the same pair of ports in the subsequent coflows. We would like to emphasize that this algorithm needs to discretize time and is based on matching source nodes to destination nodes. We select the time unit to be 1/128 second as suggested in [69] so that each port has a capacity of 1 MB per time unit. We refer to this algorithm as '<u>LP-II-GB</u>', where II stands for Interval-Indexed, and GB stands for Grouping and Backfilling.

3. Ordering-Variable-Grouping (LP-OV-GB) [71]: We implement the deterministic algorithm in [71]. Linear programming formulation is the same as LP in (3.7). Coflows are then grouped based on the optimal solution to the LP. To schedule coflows of each group, we construct a single aggregate coflow denote by D and schedule its flows to optimize its completion time. We assign transmission rate $x_{ij} = d_{ij}/W(D)$ to the flow from source node i to destination node j until its completion. Moreover, the continuous backfilling is done as follows: After assigning rates to aggregate coflow, we increase x_{ij} until either capacity of link i or link j is fully utilized. We continue until for any node, either source or destination node, the summation of rates sum to one. We also transmit flows respecting coflow order inside of each partition. When there is no more service requirement on the pair of input i and output j for coflows of current partition, we backfill (transmit) in order from the flows on the same pair of ports from the subsequent coflows. We refer to this algorithm as 'LP-OV-GB', where OV stands for ordering variables, and GB stands for Grouping and Backfilling.

4. Algorithm 3 (LP-OV-LS): We implement our algorithm as described in Algorithm 3, and refer to it as 'LP-OV-LS', where OV stands for ordering variables, and LS stands for list scheduling.

3.8.3 Evaluation Results

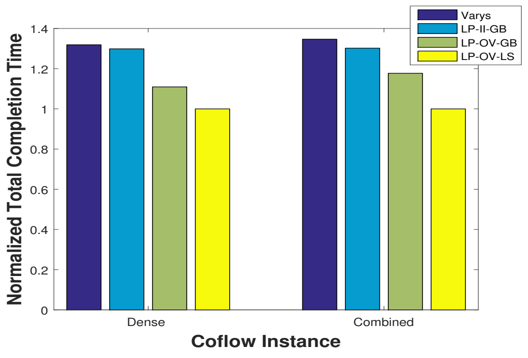
Performance of Our Algorithm. We report the ratios of total weighted completion time obtained from Algorithm 3 and the optimal value of relaxed linear program (3.7) (which is a lower bound on the optimal value of the coflow scheduling problem) to verify Theorem 2 and Corollary 1. We only present results of the simulations using the real traffic trace, with equal weights and random weights. For the case of random weights, the weight of each coflow is chosen uniformly at random from the interval $[0, 1]$. The results are more or less similar for other collections and for synthetic traffic traces and all are consistent with our theoretical results.

Table 3.2 shows the performance ratio of the deterministic algorithm for the cases of with and without release dates. All performances are within our theoretical results indicating the approximation ratio of at most 4 when all coflows release at time 0 and at most 5 when coflows have general release dates. In fact, the approximation ratios for the real traffic trace are much smaller than 4 and 5 and very close to 1.

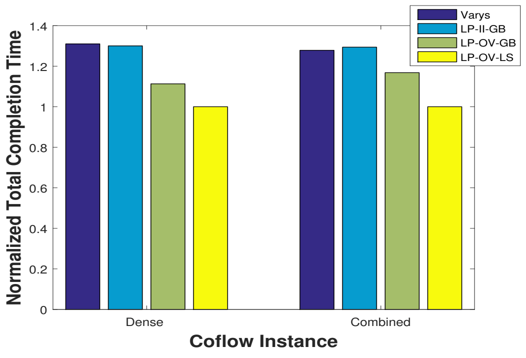
Performance Comparison with Other Algorithms. Now, we compare the performance of Algorithm 3 (LP-OV-LS) with LP-II-GB, LP-OV-GB, and Varys. We set all the weights of coflows to be equal to one.

Table 3.2: Performance ratio of Algorithm 3

Case	Equal weights	Random weights
Without release dates	1.05	1.06
With release dates	1.034	1.038



(a) All coflows release at time 0.



(b) General release dates.

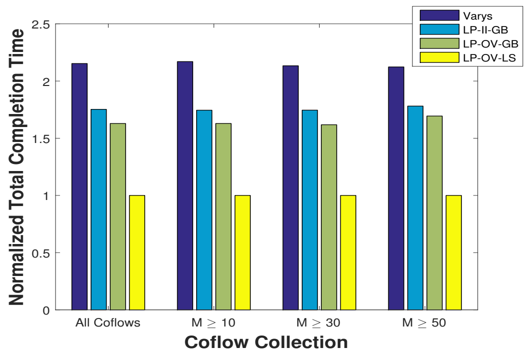
Figure 3.4: Performance of Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS for 100 random dense and combined instances, normalized with the performance of LP-OV-LS

1. Performance evaluation under synthetic traffic: For each of the two instances explained in Section 3.8.1, we randomly generate 100 different traffic traces and compute the average performance of algorithms over the traffic traces.

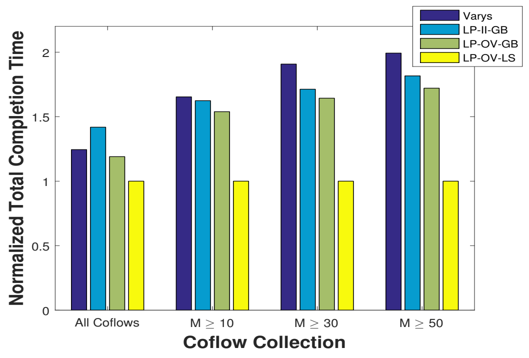
Figure 3.4a and 3.4b depict the average result of our simulations (over 100 dense and 100 combined instances) for the zero release dates and general release dates, respectively. As we see, Algorithm 3 (LP-OV-LS) outperforms Varys and LP-II-GB by almost 30%, and LP-OV-GB by almost 11% in dense instance for both general and zero release dates. In combined instance, the improvements are 35%, 30%, and 17% when all coflows are released at time 0, and 28%, 29%, and 17% for the case of general release dates over Varys, LP-II-GB, and LP-OV-GB, respectively.

This workload is more intensive in the number of non-zero flows; however, more uniform in the flow sizes and source-destination pairs in comparison to the real traffic trace. The real traffic trace (described in Section 3.8.1) contains a large number of sparse coflows; namely, about 50% of coflows have less than 10 flows. Also, it widely varies in terms of flow sizes and source-destination pairs in the network. We now present evaluation results under this traffic.

2. Performance evaluation under real traffic: We ran simulations for the four collections of



(a) All coflows release at time 0.



(b) General release dates.

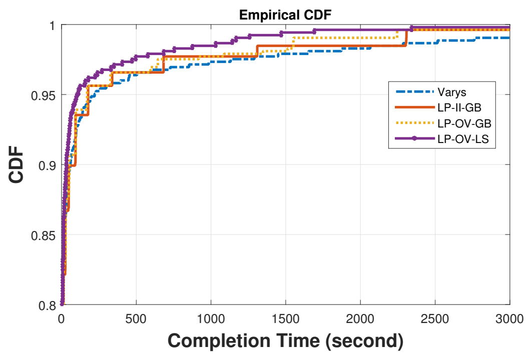
Figure 3.5: Performance of Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS, normalized with the performance of LP-OV-LS, under real traffic trace.

coflows described in Section 3.8.1. We normalize the total completion time under each algorithm by the total completion time under Algorithm 3 (LP-OV-LS).

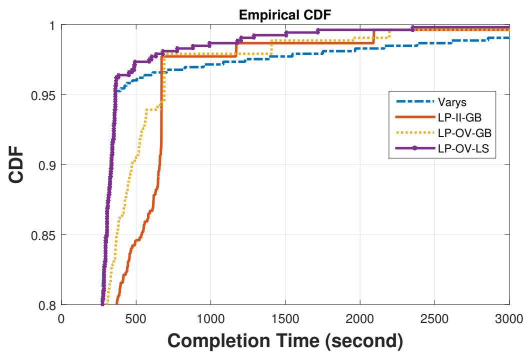
Figure 3.5a shows the performance of different algorithms for different collections of coflows when all coflows are released at time 0. LP-OV-LS outperforms Varys by almost $112 - 117\%$ in different collections. It also constantly outperforms LP-II-GB and LP-OV-GB by almost $74 - 78\%$ and $63 - 68\%$, respectively.

Figure 3.5b shows the performance of different algorithms for different collections of coflows for the case of release dates. LP-OV-LS outperforms Varys by almost 24%, 65%, 91%, and 99% for all coflows, $M \geq 10$, $M \geq 30$, $M \geq 50$, respectively. It also outperforms LP-II-GB for 40%, 62%, 71%, and 82%, and LP-OV-GB by 19%, 54%, 64%, and 73%, respectively.

Figure 3.6 depicts the CDF plots of coflow completion time for all four algorithms when all coflows are considered, for both cases of with and without release dates. Based on the plots, 95% of all coflows have completion time less than 100 seconds under our algorithm, while this is 220 seconds for Varys, when all release dates are zero. Also the CDF plots under our algorithm are quite sharp which means that the variance of completion times under our algorithm is smaller than the other algorithms.



(a) All release dates are 0.



(b) General release dates.

Figure 3.6: CDF of coflow completion time under Varys, LP-II-GB, LP-OV-GB, and LP-OV-LS for real traffic trace a) when all coflows release at time 0, b) in the case of release dates.

3.8.4 Incorporating Fairness

So far, we focused on minimizing the total weighted completion time of coflows, without considering any fairness among the rates allocated to different coflows. In this section, we propose a simple adjustment to our algorithm to provide a trade-off between fairness and optimality, and provide simulation results to study the effect of the fairness adjustment.

We use a simple metric to quantify fairness (or equivalently unfairness) among coflows. Define $p_t(k)$, the progress of coflow k by time t, to be the amount of decrease in its effective size by time t, formally,

$$p_t(k) = W(k) - W_t(k), \tag{3.19}$$

where $W(k)$ is the original effective size of coflow k (its effective size at its release date) and $W_t(k)$ is its effective size at time t after possibly partial transmission of some of its flows (recall (3.5) for the definition of coflow's effective size). Ideally, for fairness issues, we might want to have an equal progress among the coflows in the system.[2]

Hence, we use the standard deviation among progress of coflows that are in the system as our unfairness metric in rate allocation to the current coflows, i.e., the larger the standard deviation of

[2]There are other notions of fairness such as max-min fair, proportional fair, and alpha-utility fair, proposed in rate allocation for flow scheduling, e.g., see [87, 88]. The situation is more complicated in coflow scheduling, since coflows have different number of flows with different overlapping structures. Extending such notions of fairness to coflow scheduling could be an interesting future research.

progresses is, the more unfair the algorithm is. Formally, the unfairness at time t is defined as

$$SD_t = \sqrt{\frac{\sum_{k \in \mathcal{K}_t} (p_t(k))^2}{K_t} - \big(\frac{\sum_{k \in \mathcal{K}_t} p_t(k)}{K_t}\big)^2}, \tag{3.20}$$

where $\mathcal{K}_t$ is the set of coflows in the network at time t and K_t denotes its cardinality.

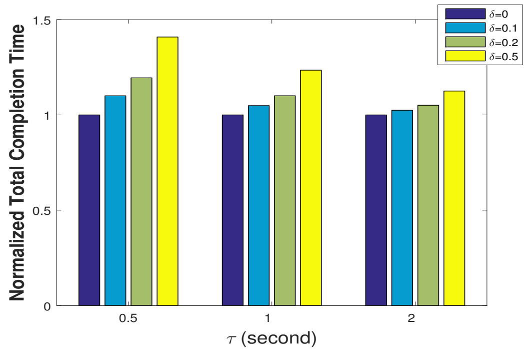
To measure unfairness throughout the entire schedule, we compute the progress of remaining coflows in the system upon departure of a coflow at time t and calculate the corresponding standard deviation according to (3.20). We then take the average of all computed standard deviations as a measure for unfairness. Based on this definition, note that a coflow that is completed earlier contributes less in the described unfairness metric because its progress is not counted in future standard deviations. Such a coflow probably has smaller effective size and its flows block less number of flows of other coflows; therefore, scheduling this coflow does not cause severe starvation for other coflows. Given this intuition, the metric captures unfairness reasonably well.

To incorporate fairness in our algorithm, we introduce two tunable parameters τ and δ ($\tau \geq \delta$) and alternate between time intervals of length τ and δ as follows. The algorithm maintains *two* lists, one is the original list in which coflows are sorted respecting inequality (3.8) and is updated upon arrival and departure of flows, and the other sorts the coflows in non-decreasing order of their progresses, as defined in (3.19). We refer to the latter list as the progress list. For a time period of length τ, we use our algorithm to schedule flows of coflows; namely, we list schedule flows according to the original list (i.e., based on optimal solution to LP). At the end of this time interval, we compute progress of coflows and update the progress list. Denote by $\bar{p}_t$ the average progress over the progress list at time t and assume that coflow k is the first coflow in the progress list. The goal of scheduling over the period δ is to decrease the gap between the progress of starved coflows and the average progress. Toward this end, we schedule the flows for a time period of length $\Delta = \min\{W_t(k), \bar{p}_t - p_t(k)\}$, where $W_t(k)$, current effective size of coflow k, is the time needed to complete coflow k ignoring other coflows in the system, and $\bar{p}_t - p_t(k)$ is the gap between its progress and the average progress. Keeping the scheduling policy simple, we use the list scheduling using the progress list for Δ amount of time. We then update the progress list, compute the average progress, current effective size of the first coflow in the progress list, and Δ, and continue in the same manner until either the total amount of time spent in this scheduling phase reaches δ or the progress of all coflows becomes equal. Afterwards, we preempt the schedule, update the original coflow list, and resume our list scheduling for another τ amount of time, and so on. Setting the parameter δ to 0 will produce our original scheduling algorithm. By choosing $\delta > 0$, we can avoid coflow starvations at the cost of an increased total completion time. Varys [8] also uses a two phase procedure, however the way that we compensate for fairness, by list scheduling based on the progress list, is different from Varys.

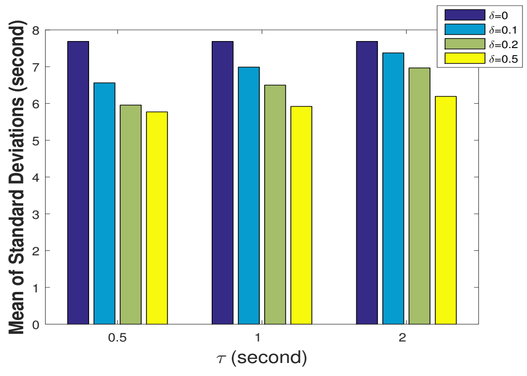
To examine the performance of the proposed scheme, we consider all coflows of the real traffic trace when they release at time zero and look at the total completion time of coflows and the unfairness metric (average of standard deviations measured upon departure of coflows) for different values of τ and δ. For practical consideration, as suggested by Varys [8], we set δ to be $O(100)$ milliseconds and T to be $O(1)$ second.
Figure 3.7a shows the total completion time for different values of δ and τ, normalized with the performance of LP-OV-LS (our original algorithm) which is when $\delta = 0$ for any value of τ. For a fixed δ, total completion time decreases as τ increases because the algorithm schedules flows based on the list that is formed to optimize total completion time for larger fraction of time. Also, fixing τ, total completion time increases as δ increases. Figure 3.7b depicts the corresponding unfairness metric for different values of δ and τ. We can see that, as δ increases, average of progress standard deviations decreases, which means that the scheduling algorithm allocates rates in a more fair manner. Moreover, fixing δ, unfairness increases as we increase τ, as expected.

3.8.5 Discussion on Algorithm's Complexity

In this section, we provide a discussion on complexity of our algorithm which is mainly determined by the step of finding appropriate ordering of coflows. The scheduling step is the simple list scheduling policy where complexity of computing the schedule– upon arrival or departure of



(a) Total coflow completion time for different values of δ and τ (both in second), normalized with the performance of LP-OV-LS ($\delta = 0$ for any τ).



(b) Average standard deviation for the progress of coflows, for different values of δ and τ (both in second).

a flow– is at most the length of the list, which is equal to the number of incomplete flows. The relaxed LP (3.7) that is used to obtain ordering of coflows has $O(K^2)$ variables and $O(K^2 + KN))$ constraints and can be solved in polynomial time, e.g. using interior point method [85]. On a desktop PC, with 8 Intel CPU core $i7 - 4790$ processors @ 3.60 GHz and 32.00 GB RAM, it took 101.93 seconds to solve the LP for the Facebook trace, when all coflows are considered for the case of general release dates. In this case, the maximum coflow completion time under our algorithm is 3492 seconds and the average completion time is 183.7 seconds. For the collection with $M \geq 50$, it took 24.40 seconds to solve the LP for the case of general release dates. In this case, the maximum coflow completion time under our algorithm is 3447 seconds and the average completion time is 194.23 seconds. We note that solving the LP can be done much faster using the powerful computing resources in today's data centers. The computation overhead as well as communication overhead (i.e., sending the rates to servers) might still be an issue for smaller coflows– the same issue as in other algorithms such as Varys [8].

3.9 NP–Completeness And Counter Example

NP-Completeness of Optimization (3.3): We first show NP-completeness of the coflow scheduling problem as formulated in optimization (3.3). This is done through reduction from the concurrent open shop problem, in which a set of K jobs and a set of N machines are given. Each job consists of some tasks where each task is associated with a size and a specific machine in which it should be processed. We convert each job to a coflow by constructing a diagonal demand matrix [69]. By this construction, the constraints (3.3d) and (3.3e) are equivalent. The optimal solution to optimization (3.3) consists of non-negative transmission rates $x_{j,j}^{k\ \star}(t)$ that sum to at most one on destination node j at each time $t \in [0, T]$. However, in the concurrent open shop problem each machine can work on one task at a time which can be translated to zero and one transmission rates in the jargon of the coflow scheduling problem. Now, we show that given an optimal solution with rates $x_{j,j}^{k\ \star}(t)$ to optimization (3.3) for the converted coflow scheduling problem, we can always transform it to a feasible solution for the original concurrent open shop problem. To do so, we consider destination node j (machine j) and start from the last flow (task) that completes on this node. If there are multiple last flows, we choose one arbitrarily. We denote by $f_{j,j}^{k\ \star}$ its optimal finishing time and by $d_{j,j}^{k}$ its size. We then set all transmission (processing) rates of this flow (task) to zero from time 0 to $f_{j,j}^{k\ \star} - d_{j,j}^{k}$, and to one from time $f_{j,j}^{k\ \star} - d_{j,j}^{k}$ to $f_{j,j}^{k\ \star}$. We adjust rates of other flows such that transmission rates sum to at most one at every time while all the flows are guaranteed to be processed before their completion time (which is given by the optimal solution). This can be easily done by increasing $x_{j,j}^{k'\ \star}(t)$ for $t \in [0, f_{j,j}^{k\ \star} - d_{j,j}^{k}]$ by $\Delta x_{j,j}^{k'}(t)$ determined as follows

$$\Delta x_{j,j}^{k'}(t) = \frac{\int_{f_{j,j}^{k\ \star} - d_{j,j}^{k}}^{f_{j,j}^{k\ \star}} x_{j,j}^{k'\ \star}(\tau) d\tau}{d_{j,j}^{k}} \times x_{j,j}^{k\ \star}(t)$$

By doing so, finishing time of the last flow does not change, and finishing time of other flows may decrease. The iterative procedure is repeated until processing rates of all flows converted to zero or one on node j. Therefore, we end up with possibly better solution in terms of total completion times of flows for node j with zero-one rates. We apply this mechanism to all nodes; hence, the total completion time of the transformed solution is as good as the optimal solution. Thus, if an algorithm can solve the coflow scheduling problem in polynomial time, it can do so for concurrent open shop problem which contradicts with its NP-completeness. This completes the argument and NP-completeness of coflow scheduling problem is concluded.

2-approximation algorithms from the concurrent open shop cannot be directly applied

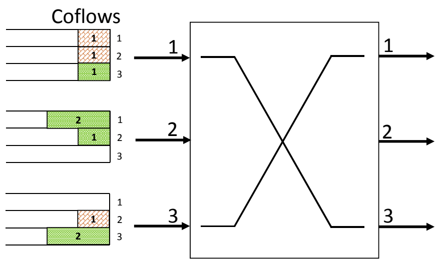
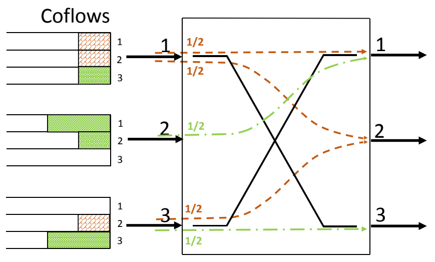
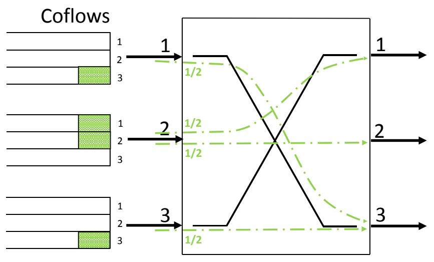


Figure 3.8: Two coflows in a 3×3 switch architecture. Flow sizes are depicted inside each flow.



(a) Transmission rates so as to complete orange coflow at time 2.



(b) Remaining flows of green coflow at time 2 and rate assignment to complete its flows at time 4.

Figure 3.9: Inaccuracy of proposed algorithm in [89] .

to coflow scheduling: As we discussed in Section 3.3, the 2-approximation algorithms for the concurrent open shop problem cannot be directly applied to achieve 2-approximation algorithms for the coflow scheduling problem. This is because given an ordering of K coflows, there does not always exist a schedule in which the first coflow completes at time $W(1)$, the second coflow completes at time $W(1,2)$, and so on, until the last coflow completes at time $W(1,\cdots,K)$ (recall Definition 3 for definition of $W(1,\cdots,k)$). We provide a counter example to show this.

Example 3 (Counter Example)**.** *Consider a* 3×3 *network with* 2 *coflows as shown in Figure 3.8. One can force the ordering algorithm to output orange coflow as the first coflow and the green coflow as the second one in the list (e.g., by means of assigning appropriate weight to coflows). To finish the first coflow (orange coflow) in* $W(1)$*, transmission rates are assigned as shown in Figure 3.9a. To avoid under-utilization of network resources, the remaining capacities are dedicated to flows of coflow* 2 *(green coflow). After* $W(1) = 2$ *units of time, coflow* 1 *completes and the remaining flows of coflow* 2 *is as shown in Figure 3.9b, therefore, one needs* 2 *more units of time* *to complete remaining flows of coflow* 2*. Hence, coflow* 2 *completes at time* $4 > W(1,2) = 3$.

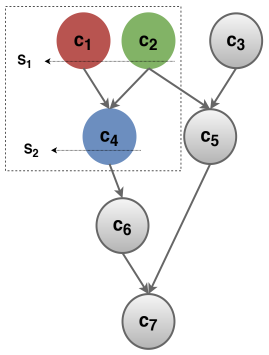
In fact, the 2-approximation algorithm in [89], for coflow scheduling when all the release dates are zero, relies on the assumption that such a schedule exists which, as we showed by the counter example, is not always true and hence the 4-approximation algorithm proposed in this chapter is the best known approximation algorithm in this case.

Chapter 4: Scheduling Coflows with Dependency Graph

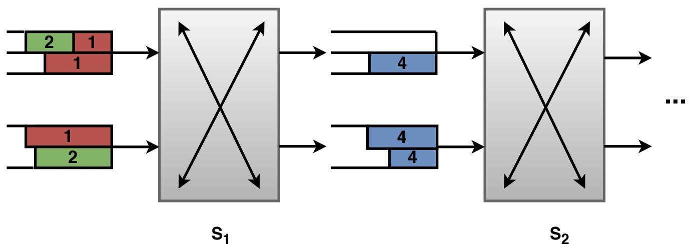
4.1 Introduction

Modern parallel computing platforms (e.g. Hadoop [90], Spark [14], Dryad [5]) have enabled processing of big data sets in data centers. Processing is typically done through multiple computation and communication stages. While a computation stage involves local operations in servers, a communication stage involves data transfer among the servers in the data center network to enable the next computation stage. Such intermediate communication stages can have a significant impact on the application latency [7]. *Coflow* is an abstraction that has been proposed to model such communication patterns [7]. Formally, a coflow is defined as a collection of flows whose completion time is determined by the last flow in the collection. For jobs with a single communication stage, minimizing the average completion times of coflows results in the job's latency improvement. However, for multi-stage jobs, minimizing the average coflow completion time might not be the right metric and might even lead to a worse performance, as it ignores the dependencies between coflows in a job [91, 9, 10].

There are two types of dependency between coflows of a multi-stage job: *Starts-After* and *Finishes-Before* [10]. A Starts-After constraint between two coflows represents an explicit barrier that the second coflow can start only after the first coflow has been completed [92]. A Finishes-Before constraint is common when pipelining is used between successive stages [5], where two dependent coflows can coexist but the second coflow cannot finish until the first coflow finishes. In this chapter we focus on scheduling coflows of multi-stage jobs with Starts-After dependency, however, our techniques and results can be easily extended to the other case. Each job is represented by a DAG (*Directed Acyclic Graph*) among its coflows that capture the (Starts-After) dependencies among the coflows. As in [78, 8, 69, 9, 91], the data center network is modeled as an



(a) A multi-stage job with 7 coflows.



(b) Flows of coflows 1, 2, and 4 and their dependencies in a 2×2 switch.

Figure 4.1: A multi-stage job in a 2×2 switch. Part of the DAG (in the dashed box) consisting of coflows 1, 2, and 4 is shown in the switch. Coflows 1 and 2 can share the network resources at the same time because they are independent (see S_1). Once all their flows are transmitted, flows of coflow 4 will be ready to be transmitted (S_2 after S_1).

$m \times m$ switch where m is the number of servers (see Section 4.2 for the formal job and data center network model). As an illustration, Figure 4.1 shows one multi-stage job in a 2×2 switch. Given a set of weights, one for each job, our goal is to minimize the total weighted completion time of jobs, where the completion time of a job is determined by the completion of the last coflow in its DAG. The weights can capture priorities for different jobs. We state the results as approximation ratios in terms of m (the number of servers), and μ (the maximum number of coflows in a job).

4.1.1 Related Work

The problem considered in this chapter can be thought of as a generalization of coflow scheduling that has been widely studied from both theory and system perspectives [8, 67, 10, 69, 76, 78, 93, 68, 94, 95]. However, there are only a few works [91, 9, 10, 96] that consider the multi-stage generalization, with only one algorithm with theoretical performance guarantee [91, 9]. Among the heuristics, Aalo [10] mainly focused on coflow scheduling problem and only provides a brief heuristic to incorporate the multi-stage case. The paper [96] proposed a two-level scheduling method based on the most-bottleneck-first heuristic to find the jobs to schedule at each round, and a weighted fair scheduling scheme for intra-job coflow scheduling.

The recent papers [91, 9] are the most relevant to our work. They consider the problem of scheduling multi-stage job (with Starts-After dependency) to minimize the total weighted job completion times and provide an LP (Linear Program)-based algorithm with $O(m)$ approximation ratio. This algorithm utilizes the technique based on ordering variables, that was also used for coflow scheduling. Their analysis for this algorithm relies on aggregating the load on all the m servers which results in the loss of $O(m)$ in the approximation ratio. *In this work, we exponentially improve this result by proposing an algorithm that achieves an approximation ratio of* $O(\mu g(m))$, where μ is the maximum number of coflows in a job, and $g(m) = \log(m)/\log(\log(m))$. Moreover, in the case that the multi-stage job's dependency graph is a rooted tree, we propose an algorithm that achieves an approximation ratio of $O(\sqrt{\mu}g(m)h(m,\mu))$, where $h(m,\mu) = \log(m\mu)/(\log(\log(m\mu))$. We would like to emphasize that the $O(m)$ approximation in [91, 9] will not improve if the graph is a rooted tree rather than a general DAG. Note that in practice, the number of coflows in a job is some constant which is much smaller than the number of servers in real-world data centers with hundreds of thousands of servers, i.e., $\mu \ll m$. Also, unlike the $O(m)$ algorithm [91, 9], both of our algorithms are completely combinatorial and do not need to solve a linear program explicitly, hence reducing the complexity. A key reason behind the performance improvement in our algorithms is that they utilize the network resources more efficiently by interleaving schedules of coflows of different jobs, unlike the $O(m)$ algorithm [91, 9] that schedules coflows one at a time.

Since we represent the dependencies between coflows of a multi-stage job with a *Directed Acyclic Graph* (DAG), DAG scheduling problem is a related line of work. In traditional DAG scheduling, each node represents a task with some processing time and an edge between two nodes indicates the tasks' dependency. There has been extensive results on DAG scheduling problem (DAG-SP) where the goal is to assign tasks to machines in order to minimize the DAG's completion time [97, 98, 99, 100, 101, 102].

There are also results on DAG-shop scheduling problem (DAG-SSP) [103, 104, 105, 106] in which, unlike the DAG-SP, the machine on which each task has to be processed is fixed and no two tasks of the same job can be processed simultaneously. Our problem of scheduling coflow DAGs is different from the aforementioned problems in several aspects: First, a node in our DAG represent a coflow which itself is a collection of data flows, each with a given pair of source-destination servers. Such couplings are fundamentally different from DAG-SP. Second, flows of the same coflow and different unrelated coflows can be scheduled at the same time, which is fundamentally different from DAG-SSP. Hence, algorithms from DAG-SP and DAG-SSP cannot be applied to our problem.

4.1.2 Main Contributions

Define $g(m) := \log(m)/\log(\log(m))$, and $h(m,\mu) := \log(m\mu)/\log(\log(m\mu))$. Our main results in this chapter can be summarized as follows.

1. We first prove that even scheduling a multi-stage job to minimize its completion time (makespan) is NP-hard. We then propose an algorithm for minimizing the time to schedule a given set of multi-stage jobs. Our algorithm runs in polynomial time and constructs a schedule in which the makespan is within $O(\mu g(m))$ of the optimal solution for the case that jobs have general DAGs, and $O(\sqrt{\mu} g(m) h(m,\mu))$ when each job is represented as a rooted tree. The algorithms rely on random delaying and merging the greedy schedules of jobs, followed by enforcing the bandwidth constraints.

2. We propose two approximation algorithms for minimizing the total weighted completion time of a given set of multi-stage jobs. For general DAGs, the approximation ratio of our algorithm is $O(\mu g(m))$. For the case of rooted trees, the ratio is improved to $O(\sqrt{\mu} g(m) h(m,\mu))$. Our algorithms are completely combinatorial and do not rely on an explicit solution of a linear program (LP), thus reducing the complexity dramatically. Our approximation algorithms are significant improvements over the LP-based $O(m)$-algorithm of [91, 9].

3. To demonstrate the gains in practice, we present extensive simulation results using real traffic traces. The results indicate that our algorithms outperform the $O(m)$-algorithm [91, 9] by up to 36% and 53% for general DAGs and rooted trees, respectively, in the same settings.

4. We illustrate the existence of instances for which the optimal makespan for a single job with a general DAG is $\Omega(\sqrt{\mu})$ factor larger than two lower bounds for the problem.

The result presented in this chapter is based on paper [107].

4.2 Model and Problem Statement

Network Model: We consider a cluster of m servers, denoted by the set $\mathcal{M}$. Each server has 2 communication links, one input and one output link with capacity (bandwidth) constraints. For simplicity, we assume all links have equal capacity and without loss of generality, we assume that all the link capacities are normalized to one. Similar to the models in [78, 8, 69, 9], we abstract out the data center network as one giant non-blocking switch. Each server in the set $\mathcal{M}$ is represented by one sender server and one receiver server. Therefore, we have an $m \times m$ switch, where the m sender (source) servers on one side, denoted by set $\mathcal{M}_S$, connected to m receiver (destination) servers on the other side, denoted by set $\mathcal{M}_R$.

Job Model: There is a collection of n multi-stage jobs, denoted by the set $\mathcal{N}$. Each job $j \in \mathcal{N}$ consists of μ_j coflows that need to be processed in a given (partial) order. Each coflow c of job j is a collection of flows denoted by an $m \times m$ demand matrix $\mathcal{D}^{(cj)}$. Every flow is a quadruple (s, r, c, j), where $s \in \mathcal{M}_S$ is its source server, $r \in \mathcal{M}_R$ is its destination server, and c and j are the coflow and the job to which it belongs. The size of flow (s, r, c, j), denoted by d_{sr}^{cj}, is the (s, r)-th element of the matrix $\mathcal{D}^{(cj)}$. For two coflows $c_1, c_2 \in j$, we say coflow c_1 *precedes* coflow c_2, and denote it by $c_1 \prec c_2$, if all flows of $\mathcal{D}^{(c_1 j)}$ should complete before we can start scheduling any flow of $\mathcal{D}^{(c_2 j)}$ (i.e., Starts-After dependency). We use a DAG G_j to represent the dependency (partial ordering) among the coflows in job j, i.e., nodes in G_j represent the coflows of job j and directed edges represent the dependency (precedence constraint) between them. We use $\mu = \max_{j \in \mathcal{N}} \mu_j$ to denote the maximum number of coflows in any job. Figure 4.1 illustrates a multi-stage job in a 2×2 switch network.

Scheduling Constraints: Without loss of generality, we assume file sizes of flows are integers and the smallest file size is at least one which is referred to as a *packet*. Scheduling decisions are restricted to such data units (packets), i.e., each sender server can send at most one packet in every time unit (time slot) and each receiver server can receive at most one packet in every time slot, and the feasible schedule at any time slot has to form a *matching* of the switch's bipartite graph. Note that the links' capacity constraints are captured by matching constraints, similarly to models in [91, 78, 69, 9]. Further, in a valid schedule, all the precedence constraints in any DAG G_i have to be respected.

Optimization Objective: A job is called completed only when all of its coflows finish their processing. Define C_{cj} to be the completion time of coflow (c, j). Then, the completion time of job j, denoted by C_j, is equal to completion time of its last coflow, i.e., $C_j = \max_{c \in j} C_{cj}$. The total time that it takes to complete all the jobs in the set $\mathcal{N}$ is called *makespan* which we denote it by $\mathcal{T}^{(\mathcal{N})}$. Note that by definition $\mathcal{T}^{(\mathcal{N})} = \max_{j \in \mathcal{N}} C_j$. Given a set of jobs, our first objective is to minimize $\mathcal{T}^{(\mathcal{N})}$. Next, given positive weights w_j, $j \in \mathcal{N}$, we consider the problem of minimizing *the sum of weighted job completion times* defined by $\sum_{j \in \mathcal{N}} w_j C_j$. The weights can capture different priority for different jobs. In the special case that all the weights are equal, the problem is equivalent to minimizing the average job completion time.

4.3 Definitions and Preliminaries

We first present a few definitions and preliminaries regarding complexity of the scheduling problem, and how to optimally schedule a single job whose graph is a path using known results.

4.3.1 Definitions

Definition 4 (Server Load and Effective Size of a Coflow)**.** *Suppose a coflow $\mathcal{D} = \left(d_{sr}\right)_{s,r=1}^{m}$ is given. Define*

$$d_s = \sum_{r \in \mathcal{M}_R} d_{sr}; \;\; d_r = \sum_{s \in \mathcal{M}_S} d_{sr}, \tag{4.1}$$

then d_s (d_r) is called the load that needs to be sent from sender server s (received at receiver server r) for coflow $\mathcal{D}$. Further, the effective size of the coflow is defined as

$$D = \max\{\max_{s \in \mathcal{M}_S} d_s, \max_{r \in \mathcal{M}_R} d_r\}. \tag{4.2}$$

Thus D is the maximum load that needs to be sent or received by a server for the coflow. Note that, due to normalized capacity constraints on links, *we need at least D time slots to process all its flows*.

Definition 5 (Aggregate Size of a Set of Coflows)**.** *Given a set of coflows, consider an aggregate coflow $\mathcal{D} = \sum_c \mathcal{D}^c$ for c's in the set. Then, aggregate size of the set is defined as the effective size of $\mathcal{D}$ based on Definition 4. Similarly, aggregate size of job j is defined as the aggregate size of its set of coflows and is denoted by Δ_j .*

Definition 6 (Size of a Directed Path and Critical Path in a Job)**.** *Given a job $j \in \mathcal{N}$ and its rooted tree G_j, size of a directed path p in G_j is defined as $T_{p,j} = \sum_{c \in p} D^{(cj)}$, where $D^{(cj)}$ is the effective size of coflow c of job j, and $c \in p$ denotes that coflow c appears in path p.*

Critical path of job j is a directed path that has the maximum size among all the directed paths in G_j. We use $T_j = \max_p T_{p,j}$ to denote its size.

Definition 7 (A Path Job)**.** *We say a job is a path job if its corresponding dependency graph is a path, i.e., there is a total ordering of its coflows according to which they should get scheduled.*

Definition 8 (A Rooted-Tree Job)**.** *We say a job is a rooted-tree job if its corresponding dependency graph is a rooted tree, i.e., it is a tree and there is a unique node called* the root *and either all the directed edges point away from this node (fan-out tree) or point toward this node (fan-in tree). For each rooted-tree job G_j, we use R_j to denote its root.*

Definition 9 (Height and Coflow Sets for a Job)**.** *Given a job $j \in \mathcal{N}$ and its graph G_j, we define H_j to be the height of G_j, i.e., the length of the longest path in G_j (in terms of number of coflows). Further, we define S_0 to denote the set of coflows with no in-edge. Similarly, define S_i, $i = 1, \ldots, H_j - 1$*

to denote the set of coflows whose longest path to some coflow of set S_0 has length i. Note that coflows in G_j are partitioned by S_is, i.e., $\cup_{i=0}^{H_j-1} S_i = G_j$ and $S_i \cap S_{i'} = \emptyset$, for $i, i' = 0, \ldots, H_j - 1$, $i \neq i'$. We refer to S_is as coflow sets of job j.

4.3.2 Complexity of Minimizing Makespan

Scheduling a multi-stage job to minimize its completion time (makespan) is NP-hard. To show this, we consider a single multi-stage job whose DAG is a rooted tree. The proof is through a reduction from preemptive makespan minimization for Flow Shop Problem (FSP) which is known to be NP-complete [108, 109, 110]. This is in contrast to traditional coflow scheduling where a single coflow can be scheduled optimally as we see in Section 4.3.3. This also shows that the known complexity results for preemptive FSP holds for single multi-stage job scheduling. For FSP, there is no algorithm with an approximation ratio less than $5/4$, unless P = NP [111].

Theorem 3. *Given a single multi-stage job represented by a rooted tree, scheduling its coflows to minimize makespan over an $m \times m$ switch is NP-hard.*

Proof. We prove the theorem using a reduction from preemptive makespan minimization for Flow Shop Problem (FSP). In FSP, there is a set of n jobs each of which consists of m tasks that need to be processed *in a given order* on m machines. Task i of job j must be scheduled on machine i for p_{ij} amount of time (all the jobs require the same order on their tasks.). Preemptive makespan minimization of FSP is known to be NP-complete [108, 110].

Consider an instance I of FSP with n jobs and m machines. We convert the makespan minimization for I to makespan minimization of an instance I' of a single multi-coflow job with a rooted tree topology. The instance I' consists of m source and m destination servers and $n \times m + 1$ coflows where each has a single flow. Further, the corresponding dependency graph of I' is a tree with a root node and n branches. The root node is a dummy coflow which has one flow of size one from source server 2 (or any other source server) to destination server 1. Each of the n branches of the tree represents a job in I and consists of m coflows. The nodes in the l-th level of the tree, $l = 1, \ldots, m - 1$ (the level of root node is zero) represent coflows that each has a single flow from source server l to destination server $l+1$ with sizes p_{lj}, $j = 1, \ldots, n$. Similarly, the nodes at level m are coflows with a single flow from source node m to destination server 1 with sizes p_{mj}, $j = 1, \ldots, n$.

If one can find the optimal makespan for the instance I' of a single multi-coflow job, the solution gives an optimal scheduling for the instance I by ignoring the first time unit that is used to schedule the dummy coflow in I'. Therefore, the theorem is proved. $\square$

Using Theorem 3 it is easy to see that minimizing makespan for multiple jobs and total weighted completion time of jobs are NP-hard.

4.3.3 Optimal Makespan for A Path Job

In this section, we first show how one can schedule a single coflow optimally and in a polynomial time using the previous results. As a result of Birkhoff-von Neumann Theorem [112], given a coflow $\mathcal{D} = \left(d_{sr}\right)_{s,r=1}^{m}$ there exists a polynomial-time algorithm which finishes processing of all the flows in an interval whose length is equal to the coflow effective size D (see Equation (4.2)).

We present one example of such an algorithm in Algorithm 4, which was proposed originally in [113], and refer to it as BNA that stands for Birkhoff-von Neumann Algorithm. BNA returns a list of matchings L and a list of times τ. To schedule flows of $\mathcal{D}$, we use each matching $L(k)$ for $\tau(k+1) - \tau(k)$ time units, for $k = 1, \ldots, |L|$.

It is immediate that the optimal makespan for a path job can be found in polynomial time, by optimally scheduling its coflows successively using BNA.

Lemma 8. *Optimal makespan for a path job j is equal to $\sum_{c=1}^{\mu_j} D^{(cj)}$ where $D^{(cj)}$ is the effective size of coflow c of job j and the corresponding schedule can be constructed in polynomial time by successively using BNA.*

We will use BNA in our algorithms in the rest of the paper.

Algorithm 4 BNA for Single Coflow Scheduling

```
Given a coflow $\mathcal{D} = \left(d_{sr}\right)_{s,r=1}^{m}$:
1. Let $L$ be the list of matchings and $\tau$ be the list of starting times for each matching. Initially, $L = \emptyset, \tau = [0]$.
2. For any $s \in \mathcal{M}_S$ and $r \in \mathcal{M}_R$, compute $d_s$, $d_r$, and $D$ according to Definition 4.
3. Find the set of tight nodes as $\Omega = (\arg\max_{s \in \mathcal{M}_s} d_s) \cup (\arg\max_{r \in \mathcal{M}_R} d_r)$.
4. Find a matching $M$ among the source and destination nodes such that all the nodes in $\Omega$ are involved.
5. Find
   $t = \min \left\{ \min_{(s,r) \in M} d_{sr}, \min_{s:(s,r) \notin M} (D - d_s), \min_{r:(s,r) \notin M} (D - d_r) \right\}$
6. Add $M$ and $t + \tau[\text{end}]$ to the lists $L$ and $\tau$, respectively.
7. Update the flow sizes as $d_{sr} \leftarrow d_{sr} - t, \ \forall (s,r) \in M$.
8. While $\mathcal{D} \neq \mathbf{0}$, repeat Steps 2 – 7.
9. Return $L$ and $\tau$.
```

4.4 Makespan Minimization for Scheduling Multiple General DAG Jobs

4.4.1 DMA (Delay-and-Merge Algorithm)

For each job j, we consider a topological sorting of nodes in G_j, i.e., we sort its coflows (nodes) such that for every precedence constraint $c_1 \prec c_2$ (directed edge $c_1 \rightarrow c_2$), coflow c_1 appears before c_2 in the ordering. This ordering is not unique and can be found in polynomial time [114]. For example, for the job in Figure 4.1a, the orderings $c_1, c_2, c_3, c_4, c_5, c_6, c_7$ and $c_2, c_3, c_1, c_5, c_4, c_6, c_7$ are both valid topological sorts. We then re-index coflows from 1 to μ_j according to this ordering.

Further, we use Δ_j to denote the maximum load that a server should send or receive considering all of job j's coflows. Formally, for job j, consider an aggregate coflow $\mathcal{D}^j = \sum_{c=1}^{\mu_j} \mathcal{D}^{(cj)}$. Then, Δ_j is the effective size of $\mathcal{D}^j$ based on Definition 4. We also use Δ to denote the maximum load a node has to send or receive *considering all the jobs*.

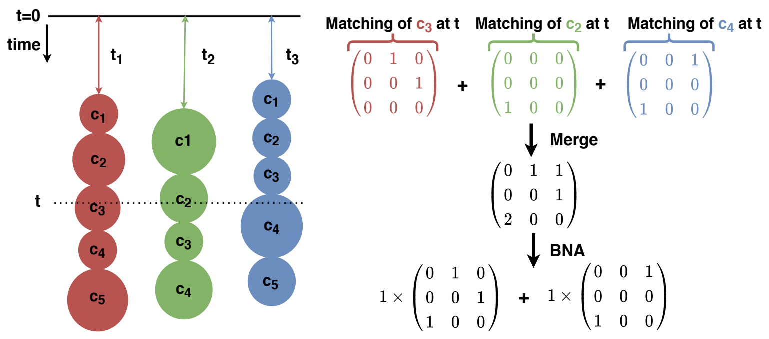


Figure 4.2: Applying DMA on 3 multi-stage jobs. On the left side, a topological ordering and a random delay for each job are computed. On the right side, the merging procedure and BNA output is shown for some time t.

Algorithm 5 (DMA) describes our algorithm for scheduling multiple general DAG jobs.

Algorithm 5 DMA for Scheduling a General DAG G_j

1. For each job j, compute a topological sorting of nodes in G_j. Then, find a feasible schedule by optimally scheduling its coflows successively using BNA, i.e., $L_{cj}, \tau_{cj} = \text{BNA}(\mathcal{D}^{(cj)})$, for coflow $c = 1, \dots, \mu_j$. We refer to these schedules as *isolated* schedules of jobs.

2. Delay each isolated schedule by a random integer time chosen uniformly in $[0, \Delta/\beta]$, for a constant $\beta > 1/e$, independently of other isolated schedules, i.e., $\tau_{cj} \leftarrow \tau_{cj} + t_j$ where t_j is the random delay of job j.

3. Greedily merge the delayed isolated schedules. I.e., for any time slot t, add corresponding matchings of different jobs.

4. Construct a feasible merged schedule. Let $\alpha_t \geq 1$ denote the maximum number of packets that a server needs to send or receive at time slot t in the merged schedule in Step 3. For each time slot t, consider an interval of length α_t, and use BNA to feasibly schedule all its packets.

Note that in DMA, in each of the isolated schedules in Step 1 all the precedence constraints among coflows are respected. However, in Step 3, the link capacity constraints may be violated. In Step 4, in the final schedule, both link capacity constraints and precedence constraints among coflows are satisfied. The parameter $\beta > 1/e$ in DMA is a constant and has no effect on the theoretical result. However, it can be used to control the range of delays in practice.

As an illustration, Figure 4.2 shows the procedure of DMA on 3 multi-stage jobs in a 3×3 switch network. On the left side, DMA computes a topological ordering for the coflows of each job and chooses a random delay for each job. The diameter of each node is proportional to the effective size of its corresponding coflow. Consider time slot t, DMA merges the matchings of coflow 3 of the red job, coflow 2 of the green job, and coflow 4 of the blue job, and inputs the result to BNA. Then, BNA computes two matchings, where each should be used for one time slot.

4.4.2 Performance Guarantee of DMA

The following theorem states the main result regarding the performance of DMA. The proof can be found in Section 4.9.1.

Theorem 4. *Given a set $\mathcal{N}$ of jobs with general DAGs, DMA runs in polynomial time and provides a feasible solution whose makespan $\mathcal{T}^{(\mathcal{N})}$ is at most $O(\mu g(m))$ of the optimal makespan with high probability, where $g(m) = \log(m)/\log(\log(m))$.*

4.4.3 De-Randomization

Step 2 of DMA involves random choices of delays. There exist well-established techniques that one can utilize to de-randomized this step and convert the algorithms to deterministic ones. For instance, one approach for selecting good delays is to cast the problem as a vector selection problem and then apply techniques developed in [115, 116, 104].

4.5 Makespan Minimization For Scheduling Multiple Rooted Tree Jobs

Now we consider the case where each job is represented by a rooted tree (Definition 8). We propose an algorithm with an improved performance guarantee compared to the case of general DAGs. We would like to emphasize that the $O(m)$ approximation algorithm [91, 9] will not be improved if the graph is a rooted tree rather than a general DAG.

4.5.1 DMA-SRT (Delay-and-Merge Algorithm For A Single Rooted Tree)

In this section, we develop an approximation algorithm for minimizing makespan of a single rooted-tree job and show that its solution is at most $O(\sqrt{\mu}\log(m\mu)/\log(\log(m\mu)))$ of the optimal

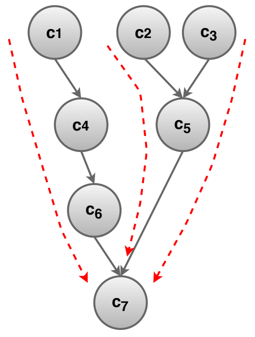


Figure 4.3: A rooted tree with 3 path sub-jobs.

makespan. Recall Definitions 8 and 9. In what follows, we assume that the rooted tree G_j has an orientation towards the root R_j (i.e. fan-in tree). For the case that edge orientations points away from the root (i.e. fan-out tree), the algorithm is similar. Recall that S_0 is the set of coflows with no in-edge in rooted tree G_j. For each coflow $c \in S_0$, we can find a directed path starting from c and ending at coflow (node) R_j. We call each of these paths a *path sub-job* of job j. We use $\mathcal{P}_j$ to denote the set of all path sub-jobs of job j. Recall that $T_{p,j}$ is the size of directed path $p \in \mathcal{P}_j$ and T_j is the size of the critical path (see Definition 6). Figure 4.3 shows a rooted tree with 3 path sub-jobs.

Algorithm 6 provides description of DMA-SRT.

Algorithm 6 DMA-SRT for Scheduling a Rooted Tree G_j

1. Find the set of path sub-jobs $\mathcal{P}_j$ of job j. For each path sub-job $p \in \mathcal{P}_j$, Choose a random integer time d_p uniformly in $[0, \Delta_j/\beta]$, for a constant $\beta > 1/e$, independent of other isolated schedules. Next, for each coflow $c \in p$, $p \in \mathcal{P}_j$, calculate the starting time of coflow c according to p, $t_{c,p} = d_p + \sum_{c'<c, c' \in p} D^{c'j}$.

2. Find the coflow sets S_i, $i = 0, \ldots, H_j - 1$ of job j according to Definition 9. For $i = 0, \ldots, H_j - 1$, and for each coflow c in S_i, find starting time of coflow c as $t_c = \min\{t_{c,p} | t_{c,p} \geq \max_{c' \in \pi_c}(t_{c'} + D^{(c'j)})\}$.

3. For each coflow c in G_j, find an optimal schedule for each coflow c using BNA, i.e., $L_c, \tau_c =$BNA$(\mathcal{D}^{(cj)})$. We refer to these schedules as *isolated* schedules. Then, delay the scheduling times by t_c, $\tau_c \leftarrow \tau_c + t_c$.

4. Follow Step 3 of DMA.

5. Follow Step 4 of DMA.

Note that the algorithm calculates the starting time of each coflow, t_c, such that all the precedence constraints of the coflow are satisfied. In other words, t_c is equal to the smallest time $t_{c,p}$ (starting time of c based on path p) that all its preceding coflows in G_j are completed. We say that c is scheduled according to p if $t_c = t_{c,p}$. Therefore, the merged schedule satisfies all the precedence constraints among coflows, although the link capacity constraints may be violated. DMA-SRT constructs a feasible merged schedule using BNA. Note that in Step 5, $\mathcal{D}$ is multiplied by l_I since each matching $L_c(i)$ runs for l_I time units in its corresponding isolated schedule. In the final schedule, both link capacity constraints and precedence constraints among coflows are satisfied.

4.5.2 Multiple Rooted Tree Jobs

Now consider the case where we have multiple jobs where each job is a rooted tree. We seek to find a feasible schedule that minimizes the time to process all the jobs (makespan). Recall that μ is the maximum number of coflows in any job. We use Δ to denote the aggregate size of coflows of *all the jobs* (Definition 5).

The scheduling algorithm is based on DMA-SRT described in Section 4.5.1. Specifically, we apply DMA-SRT to find a feasible schedule for each job in the set. Then we apply Steps 2, 3 and 4 of DMA, namely, we choose a random delay in $[0, \Delta/\beta]$ for a constant $\beta > 1/e$ for each individual schedule and delay it. Next, we merge the delayed schedules. Finally we use BNA algorithm to resolve any collisions in the merged schedule. We refer to this algorithm as DMA-RT.

4.5.3 Performance Guarantee of DMA-SRT and DMA-RT

Theorem 5. *Given a single job j with rooted tree G_j, DMA-SRT runs in polynomial time and provides a feasible schedule whose makespan C_j is at most $O(\sqrt{\mu_j}h(m, \mu_j))$ of the optimal makespan with high probability, where $h(m, \mu) = \log(m\mu)/\log(\log(m\mu))$.*

Theorem 6. *Given a set $\mathcal{N}$ of jobs, each represented as a rooted tree, DMA-RT runs in polynomial time, and achieves a solution whose makespan $\mathcal{T}^{(\mathcal{N})}$ is at most $O(\sqrt{\mu}g(m)h(m, \mu))$ of the optimal* *makespan with high probability.*

The proofs of Theorems 5 and 6 are presented in Section 4.9.2.

4.6 Total Weighted Completion Time Minimization

We are now ready to present our combinatorial approximation algorithm for minimizing the total weighted completion time of multi-stage jobs with release times. *In this section, we assume that the jobs have general DAGs, however, the results can be customized for the case that all the jobs are represented by rooted trees.* We use ρ_j to denote the release time of job j, which implies that job j is available for scheduling only after time ρ_j.

4.6.1 Job Ordering

To formulate a relaxed linear program for our problem, we note that if we ignore the precedence constraints among coflows of a job and aggregate all its coflows, we obtain a single-stage job (a coflow), and our problem is reduced to traditional coflow scheduling problem [69, 76, 78, 71].

Here, we use an LP formulation for such constructed single-stage jobs, but with an extra constraint for each job which roughly captures the barrier constraints among its coflows. Formally, for each job j, consider the aggregate coflow $\mathcal{D}^j = \sum_{c=1}^{\mu_j} \mathcal{D}^{(cj)}$. Let $\overline{\mathcal{M}} := \mathcal{M}_S \cup \mathcal{M}_R$. We use d_i^j, $i \in \overline{\mathcal{M}}$ to denote the load of coflow $\mathcal{D}^j$ on server i (see Definition 4). Recall Definition 6 and note that T_j is the lower bound on the required time to schedule multi-stage job j (in the original problem). Let $\mathcal{J}$ be any subset of jobs in $\mathcal{N}$. We formulate the following LP (Linear Program):

$$\min \sum_{j \in \mathcal{N}} w_j C_j \qquad (\mathbf{LP}) \tag{4.3a}$$

$$\sum_{j \in \mathcal{J}} d_i^j C_j \geq \frac{1}{2}\Big(\sum_{j \in \mathcal{J}} (d_i^j)^2 + (\sum_{j \in \mathcal{J}} d_i^j)^2 \Big),\ i \in \overline{\mathcal{M}}, \mathcal{J} \subseteq \mathcal{N} \tag{4.3b}$$

$$C_j \geq T_j + \rho_j,\ \ j \in \mathcal{N}. \tag{4.3c}$$

Constraints (4.3b) capture the links' capacity constraints and are used to lower-bound the completion time variables. To see this, consider a (source or destination) server i and a subset of jobs $\mathcal{J}$. For each j in $\mathcal{J}$, the completion time C_j of its aggregate coflow $\mathcal{D}^j$, has to be at least the summation of loads of coflows $\mathcal{D}^{j'}$ on server i that finish before j plus its own load on server i. Also note that for every two coflows in the set $\mathcal{J}$, one finishes before the other one. Therefore, $\sum_{j \in \mathcal{J}} d_i^j C_j \geq \sum_{j \in \mathcal{J}} d_i^j (d_i^j + \sum_{j' \in \mathcal{J}, j' \prec j} d_i^{j'})$, where $j' \prec j$ means $C_{j'} \leq C_j$. From this, Constraint (4.3b) is derived easily.

Note that this LP has *exponentially many constraints*, since we need to consider all the subsets of $\mathcal{N}$. However, we do not need to explicitly solve this LP and we only need to find an efficient ordering of jobs. To do so, we utilize the combinatorial primal-dual algorithm that first proposed in [75] and later generalized in [76] to capture constraints of the form (4.3c) for parallel scheduling problems. The algorithm builds up a permutation of the jobs in the reverse order iteratively by changing the corresponding dual variables to satisfy some dual constraint. Next, we provide the detailed explanation of the combinatorial algorithm for completeness. We then show how we use this ordering to find the actual schedule of jobs' coflows.

4.6.2 Job Ordering

In this section, we provide the detailed explanation of the combinatorial algorithm used in G-DM to find a good permutation of jobs.

Recall LP (4.3). Define $f_i(\mathcal{J})$ to be the right-hand side of Constraints (4.3b) for server i and subset of jobs $\mathcal{J}$, i.e.,

$$f_i(\mathcal{J}) = \frac{1}{2}\Big(\sum_{j \in \mathcal{J}} (d_i^j)^2 + (\sum_{j \in \mathcal{J}} d_i^j)^2 \Big). \tag{4.4}$$

We now formulate dual of LP (4.3) as follows:

$$\max \sum_{i \in \overline{\mathcal{M}}} \sum_{\mathcal{J} \subseteq \mathcal{N}} \lambda_{i,\mathcal{J}} f_i(\mathcal{J}) + \sum_{j \in \mathcal{N}} \eta_j (T_j + \rho_j) \quad \textbf{(Dual LP)} \tag{4.5a}$$

$$\sum_{i \in \overline{\mathcal{M}}} \sum_{\mathcal{J} : j \in \mathcal{J}} d_i^j \lambda_{i,\mathcal{J}} + \eta_j \leq w_j, \quad j \in \mathcal{N} \tag{4.5b}$$

$$\eta_j \geq 0, \quad j \in \mathcal{N} \tag{4.5c}$$

$$\lambda_{i,\mathcal{J}} \geq 0, \quad i \in \overline{\mathcal{M}}, \ \mathcal{J} \subseteq \mathcal{N}. \tag{4.5d}$$

The algorithm is presented in Algorithm 7. Let $\mathcal{N}'$ be the set of unscheduled jobs, initially $\mathcal{N}' = \mathcal{N}$. Also, set $\eta_j = 0$ for $j \in \mathcal{N}$. Define Λ to be the set of $\lambda_{i,\mathcal{J}}$'s that get specified in the algorithm, and initialize $\Lambda = \emptyset$ (to avoid initializing all the $\lambda_{i,\mathcal{J}} = 0$, which takes exponential amount of time) (line 1). In any iteration, let j be the unscheduled job with the greatest $T_j + \rho_j$, let ϕ be the server with the highest load and let d_ϕ be the load on server ϕ (lines 3 and 4). Now, if $T_j + \rho_j > d_\phi$, we raise the dual variable η_j until the corresponding dual constraint is tight and place job j to be the last job in the permutation (lines 5-7). However, if $T_j + \rho_j \leq d_\phi$, we choose job j' as in line 9. Then we define the dual variable $\lambda_{\phi,\mathcal{N}'}$, set it so that the dual constraint for job j' becomes tight, and place job j' to be the last in the permutation (lines 10-12).

Remark 3. *Algorithm 7 runs in $O(n(\log(n) + m))$ time where n is the number of jobs and m is the number of servers. However, the time complexity of the best known algorithm for solving the LP used in [91, 9] is $O((n^2 + m)^\omega \log((n^2 + m)/\epsilon))$, where ω is the exponent of matrix multiplication and ϵ is the relative accuracy [117, 118]. For current value of $\omega = 2.38$ [119, 120], the time complexity of Algorithm 7 is dramatically lower than the time complexity for solving the LP used in [91, 9].*

4.6.3 Grouping Jobs

Let D_j denote the maximum load that a server has to send or receive considering all coflows of the jobs up to and including job j according to the computed ordering. In other words, D_j is

Algorithm 7 Combinatorial Algorithm for Job Ordering

Given a set of multi-stage jobs $\mathcal{N}$:

```
1:  $\mathcal{N}' = \mathcal{N}$, $\eta_j = 0$ for $j \in \mathcal{N}$, $\Lambda = \emptyset$.
2:  for $k = n, n-1, ..., 1$ do
3:      $\phi(k) = \arg\max_{i \in \overline{\mathcal{M}}} d_i$
4:      $j = \arg\max_{l \in \mathcal{N}'} T_l + \rho_l$
5:      if $T_j + \rho_j > d_{\phi(k)}$ then
6:          $\eta_j = w_j - \sum_{i \in \overline{\mathcal{M}}} \sum_{\mathcal{J}, j \in \mathcal{J}} d_i^j \lambda_{i,\mathcal{J}}$.
7:          $\sigma(k) = j$.
8:      else
9:          $j' = \arg\min_{j \in \mathcal{N}'} \big(\frac{w_j - \sum_{i \in \overline{\mathcal{M}}} \sum \mathcal{J}, j \in \mathcal{J} d_i^j \lambda_{i,\mathcal{J}}}{d_{\phi(k)}^j}\big)$.
10:         $\lambda_{\phi(k),\mathcal{N}'} = \big(\frac{w_{j'} - \sum_{i \in \overline{\mathcal{M}}} \sum \mathcal{J}, j' \in \mathcal{J} d_i^{j'} \lambda_{i,\mathcal{J}}}{d_{\phi(k)}^{j'}}\big)$
11:         $\Lambda \leftarrow \Lambda \cup \{\lambda_{\phi(k),\mathcal{N}'}\}$.
12:         $\sigma(k) = j'$.
13:     end if
14:     $\mathcal{N}' \leftarrow \mathcal{N}' / \sigma(k)$.
15:     $d_i \leftarrow d_i - d_i^{\sigma(k)}, \ \forall i \in \overline{\mathcal{M}}$.
16: end for
17: Output permutation $\sigma$.
```

the effective size of an aggregate coflow constructed from coflows of the first j jobs. Recall that T_j is size of the critical path in job j (Definition 6). Define $\gamma = \min_{s,r,c,j} d_{sr}^{cj}$ which is a lower bound on the time required to process any job. Also let $T = \max_j \rho_j + \sum_{j \in N} \sum_{c \in j} \sum_{s \in M_S} \sum_{r \in M_R} d_{sr}^{cj}$. The algorithm groups jobs into B groups as follows.

Choose B to be the smallest integer such that $\gamma 2^B \geq T$, and consequently define

$$a_b = \gamma 2^b, \text{ for } b = -1, 0, 1, ..., B. \tag{4.6}$$

Then the b-th interval is defined as the interval $(a_{b-1}, a_b]$ and the group $\mathcal{J}_b$ is defined as the subset of jobs whose $T_j + \rho_j + D_j$ fall within the b-th group, i.e.,

$$\mathcal{J}_b = \{j \in N : T_j + \rho_j + D_j \in (a_{b-1}, a_b]\}; \ 0 \leq b \leq B. \tag{4.7}$$

This partition rule ensures that every job falls in some group.

4.6.4 Scheduling Each Group $\mathcal{J}_b$

To schedule jobs of each group $\mathcal{J}_b$, $b \in \{1, \cdots, B\}$, (defined by (4.7)), we use the DMA algorithm. We refer to this algorithm as G-DM algorithm which stands for *Grouping jobs, followed by Delay-and-Merge* algorithms. We summarize G-DM in Algorithm 8.

Algorithm 8 G-DM for Scheduling Multi-Stage Jobs

1. Find an efficient permutation of jobs using Algorithm 7 and re-index them.
2. Let D_j be effective size of the *aggregate* coflow constructed from coflows of the jobs up to and including job j. Also, let T_j be size of the critical path in job j.
3. Partition jobs into disjoint subsets $\mathcal{J}_b$, $b = 0, ..., B$ as in (4.7).
4. For each group $b = 1, \ldots, B$, wait until all jobs in $\mathcal{J}_b$ arrive, then apply the makespan minimization algorithm DMA to schedule them.

4.6.5 Performance Guarantee of G-DM

Recall that $g(m) = \log(m)/\log(\log(m))$, and $h(m, \mu) = \log(m\mu)/(\log(\log(m\mu))$. The following theorem states the main result regarding the performance of G-DM.

Theorem 7. *G-DM is a polynomial-time $O(\mu g(m))$-approximation algorithm for the problem of total weighted completion time minimization of multi-coflow jobs with release dates.*

For the case that we are given a set $\mathcal{N}$ of jobs, each represented as a rooted tree, we modify G-DM by using DMA-RT as the subroutine in the last step of G-DM. We denote the modified version as G-DM-RT. We then have the following Corollary.

Corollary 2. *G-DM-RT is a polynomial-time algorithm with approximation ratio $O(\sqrt{\mu} g(m) h(m, \mu))$ for minimizing the total weighted completion time of rooted-tree jobs with release times.*

The proofs can be found in Section 4.9.3.

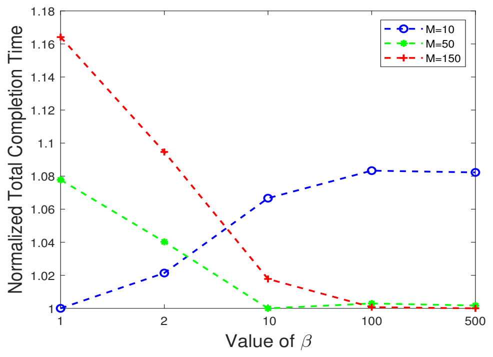


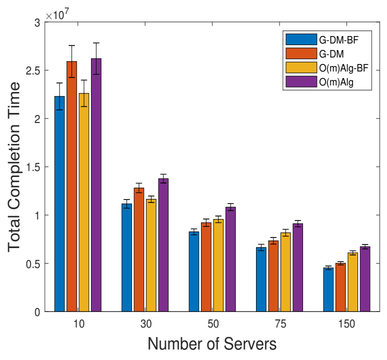
Figure 4.4: Performance of G-DM-RT for different number of servers and different values of β, and $\bar{\mu} = 5$.

4.7 Empirical Evaluation

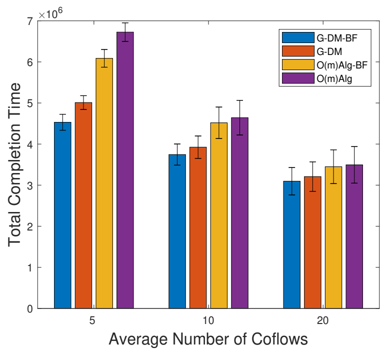
To demonstrate the gains in practice, we conducted extensive evaluations using a real workload. This workload has been widely used in coflow related research [8, 69, 9, 78]. We compared the performance of our algorithm G-DM-RT with the $O(m)$-algorithm in [91, 9] which is the previous state-of-the-art algorithm and compare its performance with that of our algorithm. In [91, 9], the authors have shown that their algorithm outperforms single-stage coflow scheduling algorithms by around 83%, and Aalo [10] by up to 33% for the case of equal weights for job (as Aalo cannot handle the weighted scenario). Hence, we only report comparison with this algorithm. The results indicate that our algorithm outperforms the $O(m)$-algorithm [91, 9] by up to 53% *in the same settings*. We also investigate the performance of the algorithms for different values of delaying parameter β, and problem size μ and m.

Workload: The workload is based on a Hive/MapReduce trace at a Facebook cluster with 150 racks, and only contains coflows information. The data set contains 267 coflows with μ_j ranging from 10 to 21170. Further, size of the smallest flow is equal to $\gamma = 1$, size of the largest flow is equal to 2472, and effective size of coflows, Δ_j, is between 5 and 232145. Finally, the maximum load a server should send or receive considering all the coflows, i.e., the effective size of the aggregate coflow, is equal to $\Delta = 440419$.

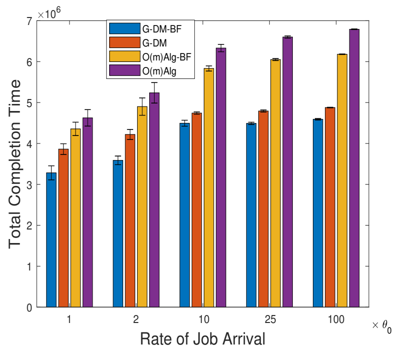
To assess performance of algorithms under different traffic intensity, we generate workloads with different number of machines (servers) by mapping flows of the original 150 racks to m



(a) Performance of G-DM and O(m)Alg with and without backfilling for different numbers of servers, and $\bar{\mu} = 5$.



(b) Performance of G-DM and O(m)Alg with and without backfilling for different average numbers of coflows per job, and $m = 150$.



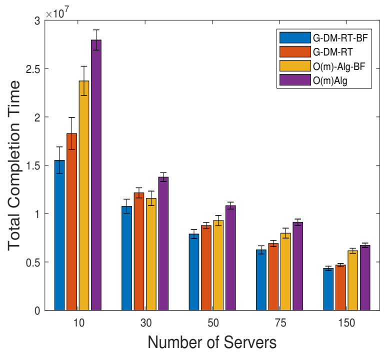
(c) Performance of G-DM and O(m)Alg with and without backfilling for different arrival rates, and $\bar{\mu} = 5$, $m = 150$.

Figure 4.5: Performance of G-DM and O(m)Alg for scheduling general DAGs with and without backfilling.

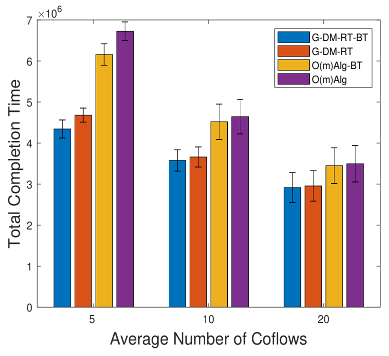
machines with various values of m. To generate multi-stage jobs, we randomly partition the coflows into multi-stage jobs that each has $\overline{\mu}$ coflows on average. To generate the corresponding rooted tree, we first generate a random graph in which probability of picking each of the edges is 0.5, and then converting it to a tree by removing its cycles. We ran the algorithms for two cases of equal weights for all jobs and randomly selected weights from interval [0, 1]. We also consider the online scenario where multi-stage jobs arrive over time and their release (arrival) times follow a Poisson process with a parameter θ.

Algorithms: We simulate our multi-stage job algorithms (referred to as G-DM and G-DM-RT) and the algorithm in [91, 9] (referred to as O(m)Alg). For each algorithm, we present two versions, one with no backfilling and one with backfilling. Backfilling is a common technique in scheduling to increase utilization of system resources by allocating the underutilized link capacities (or servers, depending on the problem) to other jobs. We apply the same backfilling strategy to both algorithms for a fair comparison. We use G-DM-BF, G-DM-RT-BF, and O(m)Alg-BF to refer to the versions of algorithms with backfilling.

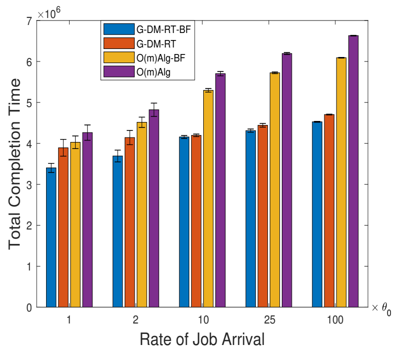
Metrics: We compare the total weighted completion times of jobs under the two algorithms for various workloads and scenarios. We present results for offline and online scenarios with equal



(a) Performance of G-DM-RT and O(m)Alg with and without backfilling for different numbers of servers, and $\bar{\mu} = 5$.



(b) Performance of G-DM-RT and O(m)Alg with and without backfilling for different average numbers of coflows per job, and $m = 150$.



(c) Performance of G-DM-RT and O(m)Alg with and without backfilling for different arrival rates, and $\bar{\mu} = 5$, $m = 150$.

Figure 4.6: Performance of G-DM-RT and O(m)Alg for scheduling rooted tree jobs with and without backfilling.

and random job weights. We also investigate the performance of the algorithms for different values of m, $\bar{\mu}$, θ.

4.7.1 Impact of Random Delays and β

The current implementations of G-DM and G-DM-RT have a random component as it uses DMA and DMA-SRT as a subroutine. To show that in practice running the algorithm *once* is sufficient to achieve a satisfactory solution, we need to show that its relative standard deviation (RSD) is small. RSD is defined as standard deviation divided by the mean (average). Hence, to analyze the effect of random delays in the performance of our algorithm, we ran it on some instances, each for 10 times. Based on our experiments, RSDs of G-DM and G-DM-RT are always less than 0.5% and RSDs of G-DM-BF and G-DM-RT-BF are always less than 0.9%, which both are very small. In the rest of simulations, we run our algorithms only once on each instance.

Furthermore, we studied the effect of parameter β (see Sections 4.4 and 4.5) on the performance of our algorithms. For each algorithm, we ran the algorithm using a wide range of β values. Based on our experiments, for smaller m (higher traffic intensity) it is better to choose a small value of β (1 or 2) to reduce the collision probability (4.15), while choosing larger β (100 or 500) for larger m helps the algorithm to use the unused capacity to schedule flows of other coflows in the system. Moreover, the amount of improvement by optimizing over β was less than 16% in all the experiments. Figure 4.4 shows the results for different values of β and m when $\bar{\mu}$ is set to 5 for G-DM-RT.

4.7.2 Evaluation Results for General GADs

Offline Setting

In the offline scenario, all the jobs are available at time 0. For each set of parameters $(m, \bar{\mu})$, we generate 10 different instances randomly and report the average and standard deviation of each algorithm's performance.

Figure 4.5a and 4.5b depict some of the results for the case that jobs have general DAGs and equal weights. Figure 4.5a shows the performance of G-DM and O(m)Alg for the case that average number of coflows per job, $\bar{\mu}$, is 5 and different number of servers. G-DM performs as well as O(m)Alg for $m = 10$. It outperforms O(m)Alg from 9% for $m = 30$ to about 36% for $m = 150$. Moreover, Figure 4.5b shows that *our algorithm outperforms O(m)Alg for all values of average coflows per job, by* 36% *to* 11%. The results for the case of random job weights are very similar and omitted.

Online Setting

For the online scenario, jobs arrives to the system according to a Poisson process with rate θ. Every time that a job arrives both G-DM (G-DM-RT) and O(m)Alg suspend the previously active jobs, update the list of jobs and their remaining demands, and reschedule them. Moreover, completion time of a job in the online scenario is measured from the time that the job arrives to the system. The job arrival rate is determined as follows: $\theta = a \times \theta_0$ for $a = \{1, 2, 10, 25, 100\}$, and $\theta_0 = \frac{\sum_j \mu_j}{\sum_j \sum_c D^{cj}}$, in which $\sum_j \mu_j$ is the total number of coflows among all jobs. The denominator, $\sum_j \sum_c D^{cj}$, is summation of coflows' effective sizes and an upper bound on the jobs' makespan.

Figure 4.5c shows the results under G-DM and O(m)Alg for the case that $m = 150$ (original data set), $\bar{\mu} = 5$, and all the jobs have equal weights. G-DM always outperforms O(m)Alg, from 20% to 36%. Furthermore, *G-DM-RT-BF always outperforms O(m)Alg-BF, by* 30% *to* 37%.

4.7.3 Evaluation Results for Rooted Trees

Now we provide the simulation results for the case that all the jobs are rooted trees.

Offline Setting

Figure 4.6a shows the performance of two algorithms for different number of servers, $\bar{\mu} = 5$, and equal weights for jobs. As we can see, G-DM-RT always outperforms O(m)Alg, for about 53% for $m = 10$ to about 46% for $m = 150$. For all values of average coflows per jobs, *our algorithm outperforms O(m)Alg, by* 46% *to* 18% as depicted in Figure 4.6b.

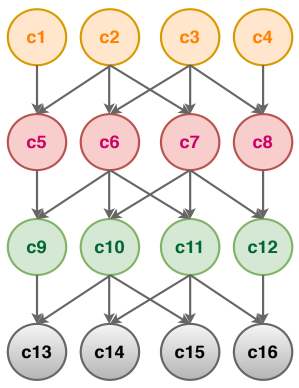
Online Setting

Figure 4.6c shows the results with and without backfilling for the case that $m = 150$ (original data set), $\bar{\mu} = 5$, and all the jobs have equal weights. G-DM-RT always outperforms O(m)Alg, from 10% to 46%. Furthermore, *G-DM-RT-BF always outperforms O(m)Alg-BF, by* 22% *to* 36%.

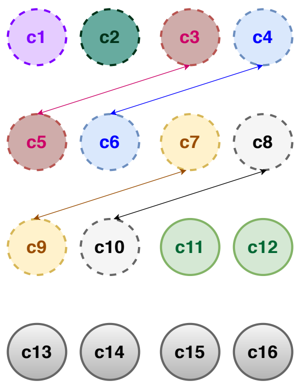
We would like to point out that, as we expect, the gain under G-DM-RT is greater than G-DM, as the former algorithm utilizes the network resources more efficiently by interleaving schedules of different coflows of the *same job* as well as interleaving schedules of coflows of *different jobs*. Furthermore, backfilling strategy generally yields a larger improvement when combined by G-DM and O(m)Alg compared to G-DM-RT, as they leave more resources unused.

4.8 Discussion on Approximation Results

An interesting research direction is to improve the approximation ratios for the algorithms. As we showed in the previous sections, once we have an algorithm for scheduling a single job whose solution is a factor η of the simple lower bounds Δ_j and T_j (Definitions 5 and 6), we can directly utilize the rest of our approach and get approximation algorithms with approximation ratio



(a) A DAG job with 16 coflows.



(b) Scheduling of coflows.

Figure 4.7: An example of a DAG with $C_{opt} = \Omega(\sqrt{\mu}(\Delta + T))$.

$O(\eta \log(m)/\log\log(m))$ for the problems of makespan minimization and total weighted completion time minimization for multiple jobs.

To improve the result for the case of general DAGs, one approach is to first consider scheduling a single job (with a general DAG), and try to generalize DMA-SRT to a general DAG by careful construction of paths in the algorithm, so that we do not need to consider all the paths in the DAG which could be exponentially many. However, even if one could show that $O(\mu_j)$ paths is sufficient to construct a feasible schedule, it is challenging to analyze the performance through computing the probability of collisions or the average number of collisions in the merged schedule as we did in proof of Lemma 11. This is due to the underlying dependency among the unrelated coflows in G_j (these are coflows among which there is no directed path in G_j, thus they can collide) which appears in the probability that a given coflow is assigned to start scheduling at a given time given that a specific $O(\mu_j)$ set of paths is generated by the algorithm.

Besides these challenges for scheduling a job with a general DAG, we can illustrate the existence of instances for which the optimal makespan is $\Omega(\sqrt{\mu_j})$ factor larger than the two simple lower bounds, Δ_j and T_j. We state this in the following lemma.

Lemma 9. *There exist arbitrary sized instances of DAG job scheduling such that its optimal makespan is* $\Omega(\sqrt{\mu_j}(\Delta_j + T_j))$.

Proof. Consider a DAG job with μ_j coflows to be scheduled in an $m \times m$ switch, with $\mu_j = (2K)^2$

for some K and $m > 2K$. Recall that T_j and Δ_j denote the size of its critical path and its aggregate size, respectively. For simplicity, we drop the subscript j. We construct the job as follows.

First, we describe the demand matrix of each coflow. For coflows $c = 1, \ldots, 2K$, each coflow has a single flow of size d from server 1 to server 2, where $2K = \sqrt{\mu}$ by assumption. These coflows are the root nodes in the job's DAG. For coflows $c = i(2K)+1, \ldots, (i+1)(2K)$, $i = 1, \ldots, (2K)-1$, each coflow has a single flow of size d from server $i+1$ to server $i+2$.

Now we specify the precedence constraints among coflows. We construct G_j such that its height is $\sqrt{\mu} = 2K$ and each of its coflow set has $\sqrt{\mu} = 2K$ coflows (see Definition 9). Consider coflow $c \in S_i$ for $i = 1, \ldots, H_j - 1$. If $i(2K) + 1 \leq c \leq (i + 1/2)(2K)$, then the parent set of coflow c is $\pi_c = \{c' | c - 2K \leq c' \leq c - K - 1\}$. If $(i + 1/2)(2K) + 1 \leq c \leq (i + 1)(2K)$, then the parent set of coflow c is $\pi_c = \{c' | c - 3K + 1 \leq c' \leq c - 2K\}$. Figure 4.7a shows an example with $\mu = 16$. For the constructed DAG, it is easy to see that $T = \Delta = 2Kd = \sqrt{\mu}d$.

Next, we specify an optimal schedule for the constructed DAG, and compute its makespan denoted by C_{opt}. We first schedule coflows $1, \ldots K$, which takes Kd amount of time. We then schedule coflows $K + 1$ and $2K + 1$ simultaneously. This is feasible since there is no precedence constraint between these two coflows, all the parents of coflow $2K + 1$ has been scheduled, and the two coflows do not share a server. Similarly, we schedule coflows $2(i - 1/2)K + c$ and $2iK + c$, for $i = 1, \ldots, 2K - 1$ and $c = 1, \ldots, K$ at the same time. Finally, we schedule the last K coflows, $c = 4K^2 - K + 1, \ldots, 4K^2$ back to back which takes Kd amount of time. For instance, consider the example of Figure 4.7. Coflow c_1 and c_2 are scheduled back to back from time 0 to $2d$. Then coflow c_3 and c_5 get scheduled from $2d$ to $3d$ and so on. Figure 4.7b shows the instance at which the first ten coflows (the coflows with dashed lines) are scheduled. The coflows with the same color (that are also linked by an arrow) have been scheduled at the same time.

By scheduling coflows in this fashion, all the precedence constraints and capacity constraints are respected. Moreover, the length of the schedule is $C_{opt} = (2K + 1)K \times d = \Omega(\mu d)$. Therefore, $C_{opt} = \Omega(\sqrt{\mu}(\Delta + T))$. □

4.9 Proofs of Main Results

In this section, we provide detailed proofs of the theorems stating performance guarantees for the proposed algorithms. Recall that $g(m) = \log(m)/\log(\log(m))$, and $h(m, \mu) = \log(m\mu)/(\log(\log(m\mu))$.

4.9.1 Proofs Related To DMA

To prove Theorem 4 we need the following lemmas.

Lemma 10. *The length of the infeasible merged schedule (Step 3) is at most* $(\mu + 1/\beta)\Delta$.

Proof. First note that the isolated schedule for job j in Step 1 spans from 0 to at most $\mu_j\Delta_j$, since the effective size of each of its coflows is at most Δ_j. By delaying the isolated schedules by at most Δ/β, length of the infeasible merged schedule is at most $\max_j(\mu_j\Delta_j) + \Delta/\beta$ which is bounded from above by $(\mu + 1/\beta)\Delta$. □

Lemma 11. *Let* $\alpha_t \geq 1$ *denote the maximum number of packets that a server needs to send or receive at time slot t in the merged schedule (Step 3). For any* $t \in [0, (\mu+1/\beta)\Delta]$, $\mathbb{E}[\alpha_t] = O(g(m))$.

Proof. Let $\overline{\mathcal{M}} := \mathcal{M}_S \cup \mathcal{M}_R$. To prove the lemma, we define random variable z_{ijt} to be 1 if some flow of job j with an end point on server i is scheduled at time slot t. Then $\alpha_t = \max_{i\in\overline{\mathcal{M}}} \sum_{j\in\mathcal{N}} z_{ijt}$. Further, note that due to the random delay of jobs' isolated schedules, variables z_{ijt}, $j \in \mathcal{N}$ are mutually independent. Let $\delta = ag(m)$ for some constant a such that $\delta > 1$. Therefore,

$$\mathbb{E}[\delta^{\alpha_t}] = \mathbb{E}[\delta^{\max_{i\in\overline{\mathcal{M}}} \sum_{j\in\mathcal{N}} z_{ijt}}] \leq \mathbb{E}\Big[\sum_{i\in\overline{\mathcal{M}}} \delta^{\sum_{j\in\mathcal{N}} z_{ijt}}\Big]. \tag{4.8}$$

Define p_{ijt} to be the probability that $z_{ijt} = 1$. By the independent property of z variables, we can write

$$\begin{aligned}
\mathbb{E}\Big[\delta^{\sum_{j\in\mathcal{N}} z_{ijt}}\Big] &= \Pi_{j\in\mathcal{N}}\mathbb{E}\Big[\delta^{z_{ijt}}\Big] \\
&= \Pi_{j\in\mathcal{N}}(p_{ijt}\delta + (1-p_{ijt})) \\
&\leq \Pi_{j\in\mathcal{N}} e^{p_{ijt}(\delta-1)} = e^{(\delta-1)\sum_{j\in\mathcal{N}} p_{ijt}} \\
&= e^{(\delta-1)\mathbb{E}[\sum_{j\in\mathcal{N}} z_{ijt}]} \leq e^{\beta(\delta-1)},
\end{aligned} \tag{4.9}$$

where the last inequality is due to $\mathbb{E}\Big[\sum_{j\in\mathcal{N}} z_{ijt}\Big] \leq \beta$. This is because by choosing delays uniformly at random, $\mathbb{E}[z_{ijt}]$ is at most the load of job j on server i divided by Δ/β, i.e., $\beta d_i^j/\Delta$, where d_i^j is the load of job j (or equivalently the aggregate coflow $\mathcal{D}^j$) on server i (see Definition 4). Thus,

$$\mathbb{E}\Big[\sum_{j\in\mathcal{N}} z_{ijt}\Big] = \sum_{j\in\mathcal{N}} \mathbb{E}[z_{ijt}] \leq \beta,$$

as $\sum_{j\in\mathcal{N}} d_i^j \leq \Delta$ by definition.

Combining Inequality (4.8) and (4.9), and by Jensen's inequality we can write,

$$\delta^{\mathbb{E}[\alpha_t]} \leq \mathbb{E}[\delta^{\alpha_t}] \leq \sum_{i\in\overline{\mathcal{M}}} e^{\beta(\delta-1)} = 2me^{\beta(\delta-1)} \tag{4.10}$$

Now, note that if we choose a sufficiently large, then $2me^{\beta(\delta-1)} \leq \delta^{\delta}$, by definition of $g(m)$. Therefore, we can conclude that $\mathbb{E}[\alpha_t] \leq \delta$, and the proof is complete. □

Lemma 12. *For any $\epsilon > 0$, the probability that the length of the final schedule (Step 4) is greater than $O(g(m))(\mu + 1/\beta)\Delta$, is less than ϵ.*

Proof. Recall that the constructed merged schedule (Step 3) spans from time 0 to at most $(\mu+1/\beta)\Delta$ due to Lemma 10. Note that, the length of the final schedule is at most $\sum_{t\in[0,(\mu+1/\beta)\Delta)} \alpha_t$. Using Lemma 11 and Markov inequality, for any $\epsilon > 0$,

$$\mathbb{P}\left(\sum_{t\in[0,(\mu+1/\beta)\Delta)} \alpha_t \geq (a/\epsilon)g(m)(\mu + 1/\beta)\Delta\right) \leq \epsilon \tag{4.11}$$

Therefore, the proof is complete.

□

Lemma 13. *Steps 3 and 4 in* DMA *can be executed in polynomial time.*

Proof. In view of Steps 3 and 4 in DMA algorithm, we may need to run BNA for $(\mu + 1/\beta)\Delta$ times. However, in the case that Δ is not polynomially bounded in m, n, and μ, we can modify the last step of DMA to ensure that it runs in polynomial time. To do so, define $H = \{\tau_{cj} | c \in G_j, j \in \mathcal{N}\}$ and $L = \{L_{cj} | c \in G_j, j \in \mathcal{N}\}$, to be the set of all scheduling times and matchings. we sort H, and let $\mathcal{I}$ be the set of time intervals created from elements of B, $\mathcal{I} = \{[h_k, h_{k+1}) | k = 1, \dots |H| - 1\}$. Thus, $\mathcal{I}$ consists of the time intervals during which the corresponding matching of every coflow is fixed.

For each interval I in $\mathcal{I}$, we merge the matchings of coflows, namely $L_{cj}(k)$'s, for which the interval I is entirely in the corresponding time interval $[\tau_{cj}(k), \tau_{cj}(k+1))$. In other words, we compute

$$\mathcal{D} = \sum_{c,j,k: I \subseteq [\tau_{cj}(k), \tau_{cj}(k+1))} L_{cj}(k).$$

Finally, for each merged matching $\mathcal{D}$, we find an optimal schedule using BNA, i.e., $L, \tau =$BNA$(l_I \times \mathcal{D})$, where l_I is length of the interval I of merged matching $\mathcal{D}$. Then we schedule demand matrix $l_I \times \mathcal{D}$ according to L and τ.

Note that whenever we run BNA, the number of elements in the list L, output of BNA, is at most m^2. This is because according to line 5 in BNA, at each iteration, t is computed such that at least one node becomes tight (i.e., it appears in the set Ω of line 3 in the next iteration) or a flow completes. Further, $|\tau| = |L| + 1$ and the last element of τ is D. Hence, in view of Steps 3 and 4 in DMA algorithm, we need to run BNA for at most $O(\mu n m^2)$ times as the number of intervals in the set $\mathcal{I}$ is $O(\mu n m^2)$. Combining this with the fact that BNA runs in polynomial time, the proof is complete. □

We are now ready to prove Theorem 4.

Proof of Theorem 4. Steps 1 and 2 in DMA can be executed in polynomial time. Combining this with Lemma 13, we can easily conclude that DMA runs in polynomial time.

Moreover, given that Δ is a lower bound for the optimal makespan, β is a constant, and Lemma 12, we conclude that makespan of the final schedule is at most $O(\mu g(m))$ of the optimal makespan with high probability. $\square$

4.9.2 Proofs Related To DMA-SRT and DMA-RT

Consider DMA-SRT. Let $\alpha_t \geq 1$ denote the maximum number of packets that a server needs to send or receive at time slot t in the infeasible merged schedule (Step 5 in DMA-SRT). To prove Theorem 5, we first state the following lemma that provides a high-probability bound on α_t.

Lemma 14. *For any $\epsilon > 0$, $\max_t \alpha_t \leq k_\epsilon \sqrt{\mu_j} h(m, \mu_j)$, with probability greater than $(1 - \epsilon)$, for a constant k_ϵ depending on ϵ, for $t \in [0, \Delta_j/\beta + T_j]$.*

Proof. To prove the lemma, let P denote the probability that any server at any time is assigned more that α packets (to be specified shortly). In what follows we first bound P_0 the probability that at least α packets are scheduled to be sent or received by a server i at time t. Note that there are at most $\binom{\Delta_j}{\alpha}$ ways to choose α packets from those that have an end point (source or destination) on server i. For packet u, the probability that it is scheduled at time t is at most $\beta|\mathcal{P}_{u,j}|/\Delta_j$, where, $\mathcal{P}_{u,j} \subseteq \mathcal{P}_j$ is the set of path-jobs containing packet u (or equivalently, the coflow to which packet u belongs.). That is because of the random uniform delay for scheduling coflows in S_0. More precisely, let $E_{u,t}$ be the event that a specific packet u is scheduled at time t and P_u be the probability that $E_{u,t}$ happens. Furthermore, let $E_{u\in p}$ denote the event that scheduling of u in the final schedule is according to the schedule of path-job p. Then,

$$\begin{aligned} P_u &= \sum_{p\in\mathcal{P}_{u,j}} \mathbb{P}\{E_{u,t}, E_{u\in p}\} \overset{(1\star)}{=} \sum_{p\in\mathcal{P}_{u,j}} \mathbb{P}\{d_p = t_p, E_{u\in p}\} \\ &= \sum_{p\in\mathcal{P}_{u,j}} \mathbb{P}\{E_{u\in p}|d_p = t_p\}\mathbb{P}\{d_p = t_p\} \end{aligned} \tag{4.12}$$

Equality (1★) is because the probability that packet u is scheduled at t *and* according to the path-job p is equal to the probability that path-job p is delayed by some specific time t_p *and* packet u is scheduled according to the path-job p. Regardless of the value of t, the probability that path-job p is delayed by t_p is either β/Δ_j or zero (if $t_p < 0$). Hence,

$$P_u \leq \frac{\beta}{\Delta_j} \sum_{p \in \mathcal{P}_{u,j}} \mathbb{P}\{E_{u\in p} | d_p = t_p\} \leq \frac{\beta |\mathcal{P}_{u,j}|}{\Delta_j} \tag{4.13}$$

Moreover, for two different packets u and v with at least a common (source or destination) server, the probability that they collide (i.e., are assigned to the same time slot) is zero if they both belong to the same coflow or same path-job, due to the feasible scheduling of each coflow and satisfaction of precedence constraints at each path-job. Otherwise, the probability that the two events $E_{u,t}$ and $E_{v,t}$ happen can be upper-bounded by multiplications of two terms of the form $\beta|\mathcal{P}_{\cdot,j}|/\Delta_j$ (using arguments similar to Equations (4.12) and (4.13)), since the random delays are chosen independently.

Therefore,

$$P_0 \leq \binom{\Delta_j}{\alpha} \Pi_{i=1}^{\alpha} P_{u_i} \leq (\frac{e\Delta_j}{\alpha})^{\alpha} (\frac{\beta}{\Delta_j})^{\alpha} \times \Pi_{i=1}^{\alpha} |\mathcal{P}_{u_i,j}| \overset{(2\star)}{\leq} (\frac{e\beta\mu_j}{\alpha^2})^{\alpha} \tag{4.14}$$

Note that the size of set $\mathcal{P}_j$ is bounded by $|S_0|$ (and therefore μ_j) as there is only one path for any coflow in S_0 to coflow R_j. Therefore, $\sum_{i=1}^{\alpha} |\mathcal{P}_{u_i,j}| \leq |\mathcal{P}_j| \leq \mu_j$. Combining this with the fact that $\Pi_{i=1}^{\alpha} |\mathcal{P}_{u_i,j}|$ is maximized when $|\mathcal{P}_{u_i,j}| = \mu_j/\alpha$, Inequality (2★) is yielded.

If we choose $\alpha = k_\epsilon \sqrt{\mu_j}$ then $P_0 \leq (m\mu_j)^{-(k_\epsilon - 1)}$. Hence, the probability that any server at any time is assigned more that α packets can be bounded by $P \leq 2m(\Delta_j + T_j)P_0 < 2m(\Delta_j + T_j)(m\mu_j)^{-(k_\epsilon-1)}$. This last step is similar to the argument in [104, 121], for job shop scheduling problem. To specify k_ϵ, note that we require P to be less than $\epsilon > 0$, which is satisfied by choosing k_ϵ as

$$k_\epsilon \geq \log_{m\mu_j} \left(\tfrac{2m(\Delta_j + T_j)}{\epsilon}\right) + 1 \tag{4.15}$$

We now need to show that k_ϵ is a constant by showing that $\Delta_j + T_j$ is polynomially bounded in m and μ_j. Let δ_j denote the maximum size of a flow in job j. Note that $\Delta_j + T_j$ is polynomially bounded in m, μ_j and δ_j. In the case that δ_j is polynomially bounded in m and μ_j, it is easy to see that by choosing k_ϵ according to (4.15), with probability $(1 - \epsilon)$, there is at most $k_\epsilon(\sqrt{\mu_j} h(m, \mu_j)$ packets on any server at any time. If δ_j is not polynomially bounded in m and μ_j, we round down each flow size d_{sr}^{cj} to the nearest multiple of $\delta_j / m^2\mu_j$ and denote it by $d_{sr}'^{cj}$. This ensures that we have at most $m^2\mu_j$ distinct values of modified flow sizes. Therefore, we can treat $d_{sr}'^{cj}$ as integers in $\{0, 1, \ldots, m^2\mu_j\}$ and trivially retrieve a schedule for $d_{sr}'^{cj}$ by rescaling. Let $\mathcal{S}'$ denote this schedule. If we increase the flow sizes from $d_{sr}'^{cj}$ to d_{sr}^{cj} in $\mathcal{S}'$ by increasing the length of the last matching that flow (s, r, c, j) is scheduled in and achieve schedule $\mathcal{S}$, we can argue that the length of $\mathcal{S}$ and $\mathcal{S}'$ differs in at most δ_j amount. This is because there are at most $m^2\mu_j$ number of flows. Thus, length of $\mathcal{S}$ is at most $(k_\epsilon + 1)\sqrt{\mu_j} h(m, \mu_j)$ as $\delta_j \leq T_j$. □

We are now ready to prove Theorem 5.

Proof of Theorem 5. It is easy to see that steps 1-3 of DMA-SRT can be done in polynomial time. By Lemma 13, Steps 4 and 5 of DMA-SRT are also executed in polynomial time. Therefore, DMA-SRT is a polynomial time algorithm.

Moreover, the completion time of each coflow is bounded by $\Delta_j/\beta + T_j$, since the maximum delay is Δ_j/β and the maximum starting time of coflow c is $T_j - D^{(cj)}$. Using Lemma 14, we conclude that the length of the final schedule is at most $O(\sqrt{\mu_j} h(m, \mu_j))(\Delta_j/\beta + T_j)$ with a high probability. Given that both Δ_j and T_j are lower bounds for the optimal makespan, the proof is complete. □

We now prove Theorem 6 regarding performance of DMA-RT.

Proof of Theorem 6. The proof is similar to proof of Theorem 4. Using DMA-SRT, completion time of job j is $O(\sqrt{\mu_j} h(m, \mu_j)) \times (\Delta_j/\beta + T_j)$. Delaying and merging these schedules and applying an argument similar to proof of Lemma 11 and 12, we can conclude that the final solution is bounded from above by $O(\sqrt{\mu}g(m)h(m,\mu)) \times (\Delta/\beta + \max_j T_j)$. Combining this with the fact that both Δ and $\max_j T_j$ are lower bounds on the optimal makespan, we can conclude the result. □

4.9.3 Proofs Related to G-DM

We use $\tilde{C}_j$ to denote the optimal solution to LP (4.3) for the completion time of job j, and use $\widetilde{\text{OPT}} = \sum_j w_j \tilde{C}_j$ to denote the corresponding objective value. Similarly we use $C_j^\star$ to denote the optimal completion time of job j in the original job scheduling problem, and use $\text{OPT} = \sum_j w_j C_j^\star$. The following lemma establishes a relation between $\widetilde{\text{OPT}}$ and OPT. To prove Theorem 7, we first show the following.

Lemma 15. *The optimal value of the LP, $\widetilde{OPT}$, is a lower bound on the optimal total weighted completion time OPT of multi-stage coflow scheduling problem, i.e., $\widetilde{OPT} \leq OPT$.*

Proof. It is easy to see that an optimal solution for the original multi-stage job scheduling problem is a feasible solution to LP (4.3) from which the lemma's statement can be concluded. □

Lemma 16. *If there is an algorithm that generates a feasible job schedule such that for any job j, $C_j^{ALG} = O(\zeta)(T_j + \rho_j + D_j)$, then $\sum_j w_j C_j^{ALG} = O(\zeta) \times OPT$, where C_j^{ALG} is completion time of job j under the algorithm.*

Proof. The proof is similar to the proofs of Lemmas 5 and 6, and Theorem 1 in [76]. We first show the following,

$$\sum_j w_j C_j^{\text{ALG}} = O(\zeta)(\sum_{i \in \overline{\mathcal{M}}} \sum_{\mathcal{J} \subseteq \mathcal{N}} \lambda_{i,\mathcal{J}} f_i(\mathcal{J}) + \sum_{j \in \mathcal{N}} \eta_j (T_j + \rho_j)), \tag{4.16}$$

for η and λ as computed in Algorithm 7 in Section 4.6.2. We would like to emphasize that the values of η and λ at the end of Algorithm 7 constitute a feasible dual solution [76]. Note that the second term in the right hand side of inequality (4.16) is the optimization objective in the Dual LP (4.5). Therefore, from weak duality (as η and λ constitute a feasible dual solution), we can conclude that $\sum_j w_j C_j^{\mathrm{ALG}} = O(\zeta) \times \mathrm{OPT}$. To show Inequality (4.16), first note that

$$w_j = \eta_j + \sum_{i \in \overline{\mathcal{M}}} \sum_{k \geq j} d_i^j \lambda_{i,k}, \tag{4.17}$$

where, with a slight abuse of notation, $\lambda_{i,k} = \lambda_{i,\mathcal{N}'}$ when $\mathcal{N}' = \{1, 2, \ldots, k\}$. Equation (4.17) is correct as job j is added to the permutation in Algorithm 7 only if Constraint (4.5b) gets tight for this job. Therefore by the lemma's assumption,

$$\sum_j w_j C_j^{\mathrm{ALG}} < O(\zeta)(\sum_j (\eta_j + \sum_{i \in \overline{\mathcal{M}}} \sum_{k \geq j} d_i^j \lambda_{i,k})(T_j + \rho_j + D_j))$$

We first bound the term $\sum_j \eta_j (T_j + \rho_j + D_j)$. Note that for every job j that has a nonzero η_j, $T_j + \rho_j > d_{\phi(j)} = D_j$. Therefore,

$$\sum_j \eta_j (T_j + \rho_j + D_j) < 2 \sum_j \eta_j (T_j + \rho_j). \tag{4.18}$$

To bound the term $\sum_j \sum_{i \in \overline{\mathcal{M}}} \sum_{k \geq j} d_i^j \lambda_{i,k} (T_j + \rho_j + D_j)$, note that for every set $\mathcal{N}' = \{1, 2, \ldots, k\}$ with nonzero $\lambda_{i,k}$, we have $T_j + \rho_j \leq d_{\phi(k)} = D_j$. Therefore,

$$\begin{aligned} &\sum_j \sum_{i \in \overline{\mathcal{M}}} \sum_{k \geq j} d_i^j \lambda_{i,k} (T_j + \rho_j + D_j) \leq 2 \sum_j \sum_{i \in \overline{\mathcal{M}}} \sum_{k \geq j} d_i^j \lambda_{i,k} D_j \\ &\leq 2 \sum_k \sum_{i \in \overline{\mathcal{M}}} \lambda_{i,k} D_k \sum_{j \leq k} d_i^j \leq 2 \sum_k \sum_{i \in \overline{\mathcal{M}}} \lambda_{i,k} D_k^2 \\ &\overset{\star}{\leq} 4 \sum_{i \in \overline{\mathcal{M}}} \sum_{\mathcal{J} \subseteq \mathcal{N}} \lambda_{i,\mathcal{J}} f_i(\mathcal{J}), \end{aligned} \tag{4.19}$$

where, Inequality ($\star$) is by (4.4) and the fact that $\lambda_{i,\mathcal{J}}$ is only nonzero for the sets of the form $\mathcal{J} = 1, 2, \ldots, k$ for some k. Combining (4.18) and (4.19), Inequality (4.8) is derived. □

Proof of Theorem 7. Recall that $\tilde{C}_j$ is the optimal completion time of job j according to the LP (4.3). Let $\widehat{C}_j$ denote the actual completion time of job j under G-DM. Also, let l_j be the index of the group to which job j belongs based on (4.7). Let j_b be the last job in group b, and T_b be the maximum size of critical paths of jobs in group b. Also let $\mathcal{T}^{(\mathcal{J}_b)}$ be the amount of time spent on processing all the jobs in $\mathcal{J}_b$. Then,

$$\begin{aligned}
\widehat{C}_j &\overset{(1\star)}{\leq} \max_{k\in b, b\leq l_j} \rho_k + \sum_{b=0}^{l_j} \mathcal{T}^{(\mathcal{J}_b)} \\
&\overset{(2\star)}{\leq} \max_{k\in b, b\leq l_j} \rho_k + O(\mu g(m)) \sum_{b=0}^{l_j} (D_{j_b} + T_b) \\
&\overset{(3\star)}{\leq} a_{l_j} + O(\mu g(m)) \sum_{b=0}^{l_j} a_b, \qquad (4.20)
\end{aligned}$$

where $g(m) = \log(m)/\log(\log(m))$. Inequality $(1\star)$ bounds the completion time of job j with sum of two terms: the first term is the maximum release time of the jobs in the first l_j groups (note that $\max_{k\in b,b\leq l_j} \rho_k$ can possibly be greater than ρ_j); The second term is the total time the algorithm spends on scheduling jobs of previous groups plus the time it spends on scheduling $\mathcal{J}_{l_j}$. Lemma 12 implies inequality $(2\star)$, and inequality $(3\star)$ follows from the fact that $\max_{k\in b,b\leq l_j} \rho_k \leq a_{l_j}$, and D_b and T_b are both bounded by a_b for every job $k \in \mathcal{J}_b$. From (4.6),

$$\textstyle\sum_{b=0}^{l_j} a_b = \gamma \sum_{b=0}^{l_j} 2^b = \gamma 2^{(l_j+1)} - 1 < 2a_{l_j}. \qquad (4.21)$$

Combining (4.20) with (4.21), and the fact that $a_{l_{j-1}} = a_{l_j}/2$,

$$\widehat{C}_j < O(\mu g(m)) a_{l_{j-1}} \overset{(4\star)}{<} O(\mu g(m))(T_j + \rho_j + D_j)$$

where $(4\star)$ is because $T_j + \rho_j + D_j$ falls in $(a_{l_j - 1}, a_{l_j}]$. This inequality combined with Lemma 16 implies that

$$\sum_j w_j \widehat{C}_j \leq O(\mu \log(m)/\log(\log(m))) \times \text{OPT}.$$

□

Chapter 5: Max-Min Fairness of Completion Times for Multi-Task Job Scheduling

5.1 Introduction

Distributed computing platforms, such as MapReduce [4], Dryad [5], Spark [14], etc., have been widely adapted for large-scale data processing in cloud and computing clusters. The data set is typically distributed among a set of servers, and processed by executing a job consisting of a set of tasks in servers. The tasks are typically processed in the servers where their input data is stored (a.k.a data locality) [4]. The collective behavior of tasks is more important than each of the tasks individually, as the job can be completed, or moved to another computation stage, *only when all of its tasks finish their processing* [4, 5, 14].

Jobs from a wide range applications and different users can coexist in the same cluster, and often have diverse tasks and processing requirements. *Efficient* and *fair* allocation of the cluster's resources among jobs is crucial to guarantee their timely completions. This has been amplified by the increasing complexity of workloads, i.e., from traditional batch jobs, to queries, graph processing, streaming, machine learning jobs, etc., that all need to share the same cluster, and often have very different latency and priority requirements. For example, analysis of a Google cluster's trace in [122] shows a diverse mix of jobs in the same cluster, ranging from latency-tolerant jobs ($\sim 24\%$) to latency-sensitive jobs ($\sim 42\%$).

Fair allocation of resources in shared clusters among applications and organizations has been studied in, e.g., [123, 124, 125], where the cluster's resources are usually divided among different applications through some notion of fairness, e.g. DRF (Dominant Resource Fair) [123]. The scheduler then manages queues of tasks for applications and schedules their tasks. For example, Hadoop [90] reserves resources by launching *containers* or *virtual machines* in servers. Each container reserves memory and CPU for *processing a task at a time*. The Hadoop scheduler uses FIFO scheduling or memory-based DRF [126]. However, these schedulers ignore the completion times of jobs and their latency requirements when allocating resources. Assigning priorities can alleviate this problem, however priorities are typically assigned to applications manually [127], and it is not clear how to assign priorities to *jobs* (and their tasks) dynamically, based on the existing jobs in the cluster and their sensitivities to latency. Further, application priority in Hadoop is supported only for FIFO scheduling [127].

Despite the vast research on scheduling algorithms (*see Related Work*), theoretical study of fairness with focus on sensitivities of jobs to latency is very limited. Moreover, prior work is mainly based on simple models that assume each job is only one task (ignoring dependency among tasks and their collective impact on the job's completion time), or tasks are processed on any server arbitrarily (ignoring data locality).

In this chapter, we consider a multi-task job scheduling model that captures such features. Each job consists of a set of tasks whose completion time is determined by the completion time of its last task. As in [128, 129], to capture latency-sensitivity, we consider a utility for each job as a function of its completion time. For example, a highly latency-sensitive job can specify a utility function that decays rapidly to zero as its completion time increases. We consider the notion of *max-min fairness* which is one of the most widely used resource allocation mechanisms [130, 131, 132]. Our objective is to schedule tasks in a way that achieves *max-min* fairness among jobs' utilities, i.e., maximize the worst utility across all the jobs, then maximize the second-worst utility without affecting the worst utility, and so on. We refer to this problem as *max-min fair scheduling*. Note that this implies that at the optimal solution, we *cannot* increase the utility of any job without hurting the utility of some job with smaller or equal utility.

We also would like to mention that the max-min fair scheduling problem for our multi-task job model is of interest from theoretical point of view. As we see later, our model can be reduced to the three scheduling problems considered in the literature to achieve max-min fairness for jobs and coflows[1] considered in [129, 133, 134]. Hence, all the three problems are at least as hard as our problem.

5.1.1 Related Work

There has been much work on fair scheduling in data centers, e.g. [125, 123, 124, 128, 135, 133, 129, 136, 137]. They mostly consider fair resource allocation to guarantee properties such as sharing-incentive among users of a shared cloud [125, 123, 124], with little focus on the sensitivity of jobs to their completion times, or consider heuristics for different notions of fairness for maximizing total utility [135], meeting deadlines [137], or fair resource assignment to each job [136]. Generating proper utility functions based on jobs' priorities and completion times was studied in [128]. In [135], a Risk-Reward heuristic was presented where scheduling decisions are made based on the cost of reallocating resources and future utility gain. The max-min fairness of job utilities were studied in [129, 133], however, their models assume each job has only one task and the cloud cluster is one large pool for each resource type. Moreover, in their solution, a job can be allocated different resource types in different unrelated time slots, as opposed to having all its required resources available at the same time. Further, despite their plausible algorithms that try to solve the problem optimally, we show in this chapter that the problems are NP-hard in a strong sense.

Our model is closely related to the *concurrent open shop* model [138, 139, 140, 75] in scheduling literature. Minimizing the (weighted) average completion time of jobs in this model has been widely studied, with several approximation algorithms in [139, 140, 75]. However, to the best of our knowledge, there is no theoretical result on max-min fairness in this model.

Max-min fair is one of the most widely used notions of fairness [130, 131, 132]. Moreover, the use of utilities and the network utility maximization for rate allocation in communication networks has been extensively studied (e.g. see [141] and references therein). However, the results cannot be extended to max-min fair job scheduling in data centers. The max-min fair optimization is not a

[1] a collection of flows whose completion time is the completion time of the last flow in the collection [7].

single-objective optimization, as we aim to optimize a vector of objective functions (utilities) in the sense of max-min fairness. Multi-objective optimization programs have been widely studied and different methods have been developed to solve these problems efficiently in special cases [142, 143, 144]. However, as we show, solving the multi-objective optimization in our setting is a hard problem (NP-hard). In [144], existence and computation of a set of equivalent weights was studied which enable the conversion of a given multi-objective optimization to a single-objective optimization. We use this method in this chapter to study the performance of one of our proposed algorithms.

We would like to mention that, there exist well-established techniques to estimate tasks' durations to the scheduler, based on the history of prior runs for recurring jobs, using tasks' peak demands, or measuring statistics from the first few tasks in each job, see [145, 146]. Hence, throughout the chapter, we assume that tasks' processing times are known.

5.1.2 Main Contributions

Our main contributions can be summarized as follows:

- **NP-Hardness of Max-Min Fair Scheduling:** We first show that it is NP-hard to optimally solve the max-min fair scheduling problem. We actually prove a stronger complexity result. Given n multi-task jobs in a cluster of machines, it is NP-hard to find a schedule in which even the first $O(n^{\epsilon}) \ll n$ number of jobs, for any $\epsilon > 0$, conform to their optimal max-min fair solution. Further, we conclude that the scheduling problems considered in [129, 133, 134] are NP-hard.

- **Approximation Algorithms:** We define two notions of approximation solutions for this problem: one based on finding a constant number of elements of the max-min fair vector, and the other based on a single-objective optimization whose solution gives the max-min fair vector. We develop scheduling algorithms, using dynamic programming and random perturbation of tasks' processing times, that provide guarantees under both notions of approximation solutions.

- **Empirical Evaluation:** We use a real traffic trace from a large Google cluster to verify that our algorithms in fact perform very well compared to other scheduling policies.

This chapter is based on the results published in the paper [147].

5.2 Model and problem statement

Cluster and Job Model: We consider a cluster of m machines, denoted by the set $M = \{1, ..., m\}$. Each machine can be thought of as a container or virtual machine [90] that can process one task at a time. There is a collection of n jobs, denoted by the set $N = \{1, ..., n\}$. Each job $j \in N$ consists of up to m different tasks that need to be processed on different machines. Each task requires a specific processing time from its corresponding machine[2]. Specifically, the task of job j on machine i, denoted by task (i, j), requires a processing time $p_{ij} \geq 0$ from machine i. For each job j, we use $M_j \subseteq M$ to denote the subset of machines that contain tasks of job j, i.e., $p_{ij} > 0$ for each $i \in M_j$. Without loss of generality, we assume processing times of all non-zero tasks are integer numbers and the smallest processing time is at least one. This can be done by defining a proper time unit and representing the tasks' processing times using integer multiples of this unit.

Tasks are independent of each other in the sense that tasks of the same job can be processed in parallel on their corresponding machines. However, a job is completed only when all of its tasks finish their processing. Define C_{ij} to be the completion time of task (i, j). Then, the completion time of job j, denoted by C_j, is given by

$$C_j = \max_{i \in M_j} C_{ij}. \tag{5.1}$$

This model is known as the *concurrent open shop* problem [138, 139, 140, 75] in scheduling literature. The total time that it takes to complete all the jobs in N is called *makespan* and is denoted by $\tau^{(N)}$. Note that by definition $\tau^{(N)} = \max_{j \in N} C_j$. It is easy to see that any valid schedule that does not leave a machine idle, unless it has completed all its corresponding tasks, achieves the

[2]In the case that a job has multiple tasks on a specific machine, we can view them as a single task with processing time equal to the cumulative original tasks' processing times.

optimal makespan which is equal to

$$\tau^{(N)} = \max_{i \in M} \sum_{j \in N} p_{ij}. \tag{5.2}$$

Max-Min Fair Objective: As in [128, 129], we assume each job j specifies a utility $U_j(C_j)$, which is a function $U_j(\cdot)$ of its completion time C_j, and captures its sensitivity to its completion time. Since each job prefers an earlier completion time, the utility function is assumed to be decreasing (with respect to the completion time). We further assume that the utility function is Lipschitz continuous (i.e., its first derivative is bounded). To define our max-min fairness, we first define the lexicographic order for two given vectors [148], as follows.

Definition 10 (Lexicographic Order)**.** *Let $X = (X_1, \cdots, X_k)$ and $Y = (Y_1, \cdots, Y_k)$ be two vectors of length k. Sort elements of X and Y in a non-decreasing order and denote the corresponding vectors by $\bar{X}$ and $\bar{Y}$, respectively. We write $X \succ Y$, and say X is lexicographically greater than Y, if $\bar{X}_i > \bar{Y}_i$ for the first i that $\bar{X}_i$ and $\bar{Y}_i$ differ. Consequently, we write $X \succeq Y$ and say vector X is not lexicographically smaller than Y if $\bar{X} = \bar{Y}$ or $X \succ Y$.*

Our objective is to schedule jobs (their tasks) in a way that achieves the max-min fairness across the vector of jobs' utilities. In other words, we wish to maximize the worst (minimum) job utility across all the jobs, and then sequentially maximize the next-worst utility without affecting the previous-worst utility, and so on. Formally, let $C = (C_1, \cdots, C_N)$ be the vector of completion times of jobs in set N and define $U(C) = (U_1(C_1), \cdots, U_N(C_N))$. We seek to schedule jobs in a way that the optimal completion time vector, denoted by $C^\star$, has the property that the vector $U(C^\star)$ is lexicographically greater than $U(C)$ for any other valid scheduling of jobs with completion time vector C, i.e., $U(C^\star) \succeq U(C)$. Note that by Definition 10, the sorted optimal vector $\bar{U}(C^\star)$ is unique.

Preemptive vs Non-preemptive Scheduling: The scheduling algorithm could be preemptive or non-preemptive. In a non-preemptive algorithm, a task cannot be preempted once it starts its processing on its corresponding machine, while in a preemptive algorithm, a task may be preempted and resumed later on the same machine.

5.3 Lexicographic Max-Min Fair Schedule and NP-hardness

In this section, we first characterize the structure of optimal schedules for max-min fair problem. Then we show a strong result regarding NP-hardness of finding the optimal schedule.

5.3.1 Structure of Optimal Schedule

In a non-preemptive schedule (Section 5.2), tasks on each machine are processed according to some order. We say a task is at position l, $l = 1, \cdots, n$, on machine i if it is the l-th task that is completed in machine i. Hence, to fully describe a non-preemptive schedule, it is sufficient to specify a permutation π_i for each machine i, $i \in M$, as formally defined below.

Definition 11 (Permutation of Tasks on Machine i)**.** *Given a set of jobs $N = \{1, 2, \ldots, n\}$ and a valid non-preemptive schedule on machine i, a permutation $\pi_i : N \to \{1, 2, \ldots, n\}$ is a one-to-one mapping of jobs to positions $\{1, 2, \ldots, n\}$ according to which their tasks on machine i are completed.*

Hence $\pi_i(j)$ determines the position of job j's task on machine i. Note that some jobs might not have any tasks on machine i. For these jobs, we consider tasks with zero processing time on machine i. These zero-processing tasks do not contribute to the completion times of jobs and their utilities; nevertheless, including them in Definition 11 will make the future arguments easier. The following theorem characterizes the structure of optimal solution.

Theorem 8. *Any optimal schedule for max-min fair problem can be converted to another optimal schedule in which all the tasks are scheduled in a non-preemptive fashion, according to the same permutation on all the machines.*

Proof Overview. Given any optimal schedule, we construct a non-preemptive schedule, with identical permutation for all machines: Starting from the last job (the job with the largest completion time) in the given solution, we move all its tasks to the end of the schedule in their corresponding machines, and sequentially do this for all the jobs. We omit the details. $\square$

Hence, by Theorem 8, in order to find an optimal solution, it is sufficient to only consider non-preemptive schedules with the same permutation of jobs $\pi_i = \pi$ on all machines $i \in M$.

5.3.2 NP-Hardness

Next, we show that finding an optimal solution to the max-min fair scheduling is NP-hard. In fact, we prove a stronger complexity result. Before stating the result, we make the following definition.

Definition 12 ($f(n)$-max-min fair)**.** *Let $\bar{U}(C)$ denote the sorted utility vector corresponding to completion time vector C. We say a solution C is $f(n)$-max-min fair, if the first $f(n)$ elements of $\bar{U}(C)$ match the first $f(n)$ elements of $\bar{U}(C^\star)$ where $C^\star$ is completion time vector for some optimal solution.*

Consider any increasing function $f(n) \leq n$ for which $f^{-1}(n)$ is bounded by a polynomial in n. We show that it is NP-hard to find a schedule (or equivalently a permutation of jobs by Theorem 8) for which the first $f(n)$ elements of the sorted utility vector matches the first $f(n)$ elements of the sorted max-min fair utility vector. We state the result in the following theorem.

Theorem 9. *Given a set of machines $M = \{1, ..., m\}$ and a set of jobs $N = \{1, ..., n\}$, scheduling jobs to achieve $f(n)$-max-min utility optimal is NP-hard, for any increasing function $f(n)$ for which $f^{-1}(n)$ is polynomially bounded in n. The result holds even if all the utility functions are the same, i.e., $U_j(C_j) = U(C_j),\ \forall j \in N$.*

For instance, Theorem 9 holds for any sublinear function $f(n) = n^\epsilon$, for any $\epsilon \in (0, 1]$, but not for $f(n) = \log(n)$.

In the case of identical utility functions, $U_j(C_j) = U(C_j),\ \forall j \in N$, it is easy to observe that "max-min" fairness among utilities is equivalent to "min-max" among the completion times. In the latter problem, we minimize the largest completion time across the jobs, and then successively minimize the next largest completion time as long as it does not affect the previous largest completion time, and so on. We formally state this fact in the following lemma.

Lemma 17. *In the case that $U_j(C_j) = U(C_j)$, $\forall j \in N$, max-min fairness among utilities is equivalent to min-max of completion times.*

Proof. Given that the utility function $U(.)$ is not increasing, the result is immediate. □

Proof of Theorem 9. We prove the theorem for the special case when all jobs' utility functions are the same. This implies NP-hardness of the problem for general cases with any non-increasing utility functions. To prove this, we use a reduction from the *Minimum Vertex Cover Problem* which is known to be NP-hard [149]. An instance $\mathcal{I}$ of Minimum Vertex Cover Problem is given by a graph $G = (V, E)$, where the goal is to find a *minimum cardinality* set of vertices $W \subset V$ such that each edge $e \in E$ is incident to at least one vertex of W. We use $\mathrm{VC}(G)$ to denote the cardinality of W. We map this instance to an instance $\mathcal{I}'$ of the problem of $f(n)$-min-max completion times using a polynomial time procedure.

Instance $\mathcal{I}'$ has $m = |E| + n'$ machines, one for each edge $e \in E$, plus n' extra machines to be specified shortly, and $n = |V| + n'$ jobs, one for each vertex $v \in V$ and extra n' jobs. Let d_v denote the degree of vertex v in G. For each vertex $v \in V$, we consider a job $j(v)$ consisting of d_v tasks $(j(v), e)$, such that $p_{j(v)e} = 1$ if edge $e \in E$ is incident to v, and 0 otherwise. We refer to these jobs as *vertex jobs*. The remaining n' jobs, each has a unit-sized task on one of the last n' machines, such that each of the n' machines only has one task to process. We refer to these jobs as *dummy jobs*. We choose $n' = f^{-1}(|V|) - |V|$. Note that $f(|V|) \leq |V|$ and f is an increasing function (and so is f^{-1}), therefore, $n' \geq 0$. At the end of this construction, each machine has either 1 or 2 tasks to process; hence, all the jobs can be scheduled in two time slots. Consider a schedule with the following properties: (1) it finishes all the jobs using two time slots, (2) all the n' dummy jobs are completed in the first time slot Note that the set of tasks completed in the second time slot belong to a set of vertex jobs. This set of vertex jobs creates a vertex cover for G, because each of the first

$|E|$ machines has to process some task from these jobs in the second time slot.

Note that by the choice of n',

$$f(n) = f(|V| + n') = f(f^{-1}(|V|)) = |V| > \mathrm{VC}(G). \tag{5.3}$$

Out of the first $f(n) = V$ jobs in the sorted completion time vector, some jobs have completion time equal to 2 and some jobs have completion time equal to 1. To find the $f(n)$-min-max vector, we therefore need to minimize the number of jobs completed in the second time slot, which is equivalent to finding the minimum vertex cover of G. Note that the remaining jobs correspond to an independent set in graph G, and hence all their tasks can be scheduled in the first time slot. However, it is NP-hard to find the minimum vertex cover of graph $G = (V, E)$ [149]. □

As a result of Theorem 9, we can conclude that the max-min fairness problem for single-task jobs considered in [129] (see Section 5.1.1 for more details) and the max-min fairness scheduling of coflows considered in [134] are both NP-hard problems, that were not shown before. The proof is based on reduction of our problem to these scheduling problems. The details are omitted.

Corollary 3. *The max-min fair scheduling problems considered in [129, 133, 134] are NP-hard.*

5.4 Defining Approximation Solutions

In single-objective optimization, in case the problem is NP-hard, we try to find approximation algorithms, which run in polynomial time, and can return a solution with provable guarantee on its distance from the optimal solution (e.g., approximation ratio) [150]. However, the optimization problem in our setting is *not* a single-objective optimization, as we aim to optimize a vector of objective functions in the sense of max-min fairness. Consequently, given that finding the optimal vector solution to our problem is NP-hard (Theorem 9), it is not clear how to define the approximation algorithms in our setting. In this section, we describe two possible definitions for approximation solutions. We focus on the case that all jobs' utility functions are the same, i.e., $U_j(C_j) = U(C_j)$, $j \in N$. Recall that by Lemma 17, this is equivalent to the problem of min-max of completion times, which is still NP-hard by Theorem 9. In Section 5.6, we discuss possible extensions to unequal utility functions.

5.4.1 **k**-Min-Max Fair Approximation

A natural way of extending the idea of approximation ratio is through αn-min-max, for some $\alpha < 1$, based on $f(n) = \alpha n$ in Definition 12. We can attempt to find an approximate algorithm (schedule) such that the first αn elements of its corresponding sorted completion time vector matches the first αn elements of the sorted min-max vector. However, Theorem 9 implies that even finding such a schedule is NP-hard for any constant $\alpha > 0$. Therefore, we ask for less, and consider *finding the first k elements of the sorted optimal vector, for a fixed constant $k < n$.*

5.4.2 Single-Objective Approximation

The second approach could be to formulate a single-objective optimization whose optimal solution coincides with the min-max vector. We can then use this single-objective optimization to measure the quality of an approximation solution to the min-max problem. We describe one such formulation based on an integer program.

An Equivalent Integer Program (IP). We formulate an Integer Program based on minimization of the total weighted completion times of jobs. In traditional minimization of total weighted completion times [139, 140, 75], each job j has a positive fixed weight w_j and the objective is to minimize $\sum_{j \in N} w_j C_j$. The optimization that we consider here is different as the weights of jobs are not fixed in advance and depend on their positions in permutation. Formally, for any position $l \in \{1, 2, \ldots, n\}$ and any job $j \in N$, we define a binary variable x_{lj} which is 1 if *job j is the l-th job to complete in the schedule*, and 0 otherwise. In view of Definition 11, $x_{lj} = 1$ is equivalent to having $\pi(j) = l$, when $\pi_i(j) = \pi(j)$ for all $i \in M$. We refer to $\{x_{lj}\}$ as permutation variables. Each position $l \in \{1, 2, \ldots, n\}$ is assigned a non-negative weight w_l. Define $C^{(l)}$ to be the completion time of the l-th job completed in the schedule

$$\textbf{(IP)}\ \min\ \sum_{l=1}^{n} w_l C^{(l)} \tag{5.4a}$$

$$C^{(l)} \geq \sum_{s=1}^{l} \sum_{j \in N} p_{ij} x_{sj},\ i \in M,\ 1 \leq l \leq n \tag{5.4b}$$

$$\sum_{l=1}^{n} x_{lj} = 1,\ j \in N \tag{5.4c}$$

$$\sum_{j \in N} x_{lj} = 1,\ 1 \leq l \leq n \tag{5.4d}$$

$$x_{lj} \in \{0, 1\},\ 1 \leq l \leq n,\ j \in N \tag{5.4e}$$

Constraint (5.4b) is based on the definition of permutation variables and the fact that the completion time of the l-th job is greater than completion time of its task on any machine i. Constraints (5.4c) and (5.4d), capture the requirement that each job is assigned to a position, and each position is assigned to a job, respectively. Let $C^{\star(l)}$ denote the value of completion time of the l-th job in an optimal solution to (IP). Observe that by the minimization objective, for any job there is a machine for which Constraint (5.4b) turns to equality at the optimal solution. Let $i^\star$ denote the machine for which $C^{\star(l-1)} = \sum_{s=1}^{l-1} \sum_{j \in N} p_{i^\star j} x^\star_{sj}$. Then, as a result of Constraint (5.4b) on machine $i^\star$ for the l-th job we have,

$$C^{\star(l)} \geq \sum_{s=1}^{l} \sum_{j \in N} p_{i^\star j} x^\star_{sj} \geq \sum_{s=1}^{l-1} \sum_{j \in N} p_{i^\star j} x^\star_{sj} = C^{\star(l-1)}.$$

This implies that $C^{\star(1)} \leq \cdots \leq C^{\star(n)}$, i.e. the values of $C^{\star(l)}$ are consistent with our definition of jobs' positions $l = 1, \cdots, n$. However, since a job l with no task on a machine i is assumed to have a task with zero processing time on that machine, and Constraint (5.4b) is on all machines $i \in M$, the completion time of the job may be dominated by one of its zero-processing tasks. This can result in a larger value for $C^{\star(l)}$ than the *actual* completion time of the l-th job in the schedule according to (5.1). We need to show that $C^{\star(l)}$ is indeed the completion time of the l-th job in the schedule.

Lemma 18. *For any job h and its corresponding position l (i.e., $x^\star_{lh} = 1$) in an optimal solution to IP (5.4),*

$$C_h = C^{\star(l)} = \sum_{s=1}^{l} \sum_{j \in N} p_{i^\star j} x^\star_{sj}, \textit{ for some } i^\star \in M_h.$$

Therefore, $C^{\star(l)}$ is indeed the completion time of job h in the schedule corresponding to optimal permutation variables $x^\star_{lj}$ (or its corresponding job permutation $\pi^\star$).

The proof of Lemma 18 is based on a contradiction argument and optimality of $C^\star$.

Proof. For purpose of contradiction, assume that the lemma statement does not hold for some l, i.e. completion time of l-th job, job h, happens at machine $i^\star$ for which $p_{i^\star h} = 0$. In other words, $x^\star_{lh} = 1$, $C^{\star(l)} = \sum_{s=1}^{l} \sum_{j \in N} p_{i^\star j} x^\star_{sj}$, and $i^\star \in M \setminus M_h$. We further assume that there is no machine in M_h at which completion time of job h happens and $i^\star$ is unique. Denote by m the machine in M_h that task p_{hm} has the maximum completion time $C^\star_{mh}$, i.e., $m = \arg\max_{i \in M_h} \sum_{s=1}^{l} \sum_{j \in N} p_{ij} x^\star_{sj}$. Therefore, $C^\star_{mh} < C^\star_h = C^{\star(l)}$.
Moreover, let $l' < l$ denotes the largest position of some job, say k, whose task at machine $i^\star$ is non-zero, i.e., $x^\star_{l'k} = 1$ and $p_{i^\star k} > 0$. Then the following inequality holds.

$$C^{\star(l')} \geq \sum_{s=1}^{l'} \sum_{j \in N} p_{i^\star j} x^\star_{sj} = \sum_{s=1}^{l} \sum_{j \in N} p_{i^\star j} x^\star_{sj} = C^{\star(l)},$$

Given that $C^{\star(l')} \leq C^{\star(l)}$ (by optimality of the solution), we conclude that $C^{\star(l')} = C^{\star(l)}$. By changing the positions of jobs h and k, and using C for the new completion times, we have the following:

$$C^{(l')} = C_h \leq \max\{C^\star_{mh}, C^{\star(l')} - p_{i^\star k}\} < C^\star_h = C^{\star(l)}.$$

Furthermore, completion time of job k is equal to $C^{(l)} = C_k = C^{\star(l')}$, therefore, we can decrease the objective function of IP (5.4) using the updated permutation. This suggests that the solution is not optimal which is a contradiction.

In the case that $i^\star$ is not unique, we denote the set of such machines by I. We then choose l' to be the largest position $< l$ of some job, say k, whose task at machine $i^\star \in I$ is non-zero, i.e., $p_{i^\star k} > 0$. The rest of the proof is similar. This completes the proof. □

Next, we show that by an appropriate choice of weights w_l, $l = 1, 2, \ldots, n$, we can force the optimal solution to IP (5.4) to coincide with the optimal min-max vector of completion times. Recall the definition of $\tau^{(N)}$ in (5.2). The following lemma states the result for non-trivial instances of min-max problem.

Lemma 19. *Let $w_0 = (\tau^{(N)})^n$ and assume that $\tau^{(N)} \geq 2$ and $n \geq 3$. The optimal solution to IP (5.4) is an optimal solution for min-max problem if we set $w_l = w_0^l$.*

Proof Overview. Consider an optimal solution $C^{\star(l)}$, $l = 1, \cdots, n$, to IP (5.4), and let $\tilde{C}^{(l)}$ be the completion time of the l-th job in a min-max solution. The proof is by contradiction. Suppose $\{C^{\star(l)}\}_{l=1}^n$ is not a min-max optimal solution. Then, it follows that there must exist some position l, $1 \leq l \leq n$, for which the following relation holds,

$$\begin{aligned} &C^{\star(l')} = \tilde{C}^{(l')} \ \forall l' > l, \\ &C^{\star(l')} > \tilde{C}^{(l)} \text{ for } l' = l, \\ &C^{\star(l')} < \tilde{C}^{(l')} \ \forall l' < l. \end{aligned}$$

We then proceed to show that by the choice of weights as in the lemma's statement, even if the completion time of the l-th job, $C^{\star(l)}$ is greater than $\tilde{C}^{(l)}$ by only *one* time unit, we get $\sum_{l'=1}^{l} w_{l'} C^{\star(l')} > \sum_{l'=1}^{l} w_{l'} \tilde{C}^{(l')}$, which contradicts the optimality of $\{C^{\star(l)}\}_{l=1}^n$ for IP (5.4). We omit the details. □

Note that the total number of bits required to represent the weights in Lemma 19 is polynomially bounded in the problem input. Specifically, the number of bits required to represent the largest weight w_n is $O(n^2 \log \tau^{(N)})$, therefore we need at most $O(n^3 \log \tau^{(N)})$ bits to represent all the weights.

5.5 Approximation Algorithms for Equal Utility Functions

In this section, we consider the case where all jobs' utility functions are the same. Before presenting our scheduling algorithms, we describe a set of permutations that contains an optimal schedule. Recall that for each job $j \in N$, M_j denotes the set of machines for which $p_{ij} > 0$.

Lemma 20. *Consider the problem of finding the optimal min-max solution of jobs' completion times. For any two jobs h and k, 1) If $p_{i,h} \leq p_{i,k}, \forall i \in M_h \cap M_k$, then there is an optimal schedule that job h precedes job k in the permutation. 2) If $p_{i,h} = p_{i,k} = p, \forall i \in M_h \cap M_k$, then there is an optimal schedule that jobs h and k are adjacent in the permutation.*

Proof. The proofs of both statements are based on exchange arguments. We omit the proof of the first statement. For proof of the second statement, consider an optimal solution with the same job permutation on all the machines (Recall Theorem 8). Assume that there are $R > 0$ jobs between job h and k in the permutation. Let denote by C_h, C_k, and C_r for $r = 1, \ldots, R$ completion times of job h, completion time of job k, and completion time of the r-th job that is between jobs h and k, respectively. Therefore, the optimal permutation is $\pi^\star = (\ldots, h, 1, 2, \ldots, R, k, \ldots)$ and the optimal completion time vector can be written as

$$[\ldots, C_h, C_1, C_2, \ldots, C_R, C_k, \ldots]^T. \tag{5.5}$$

We now show that one of the two following permutations in which jobs k and h are incident is also an optimal permutation.

$$\pi_1 = (\ldots, h, k, 1, 2, \ldots, R, \ldots)$$
$$\pi_2 = (\ldots, 1, 2, \ldots, R, k, h, \ldots).$$

Assume that π_1 is not optimal. The corresponding sorted completion time vector for this permutation is

$$[\dots, C_h, C_h + p, C_1 + p, C_2 + p, \dots, C_R + p, \dots]^T.$$

Note that $C_R + p = C_k$. Since this permutation is not optimal, there exists some job with index x, $1 \le x \le R$ with the following property.

$$\begin{aligned} &C_{x-1} + p > C_x \\ &C_{l-1} + p = C_l \ \ \forall l > x. \end{aligned} \tag{5.6}$$

In the case that $x = 1$, job 0 refers to job h. Now consider the other permutation π_2. The corresponding sorted completion time vector for this permutation is

$$[\dots, C_1 - p, C_2 - p, \dots, C_R - p, C_R, C_R + p, \dots]^T,$$

in which $C_R + p = C_k$. Value of the l-th element in this vector is $C_l - p$ for $1 \le l \le R$. Comparing this vector with the optimal vector in (5.5), value of the last two elements are equal. Also, the l-th element in the optimal vector is C_{l-1}, with $C_0 = C_h$ and $C_{R+1} = C_k$. By property (5.6) we know that $C_{x-1} + p > C_x$ which implies that $C_x - p < C_{x-1}$ for some x and $C_l = C_{l-1} + p$ for $l > x$. This means that π_2 is strictly better than $\pi^\star$ which contradicts with optimality of the latter permutation. This implies that a permutation in which job h and k are incident is also optimal. $\square$

We use Lemma 20 later in this section to augment the solution of an algorithm.

5.5.1 **k**-Max-Min Scheduling Algorithm

We aim to find a k-min-max fair schedule as defined in Section 5.4.1. This is equivalent to finding the last k jobs in the corresponding optimal permutation. Algorithm 9 gives a description of our algorithm. It is based on dynamic programming and starts by finding the last job and moves backward to find the last k jobs in the optimal permutation.

Let π_1 be the output of Algorithm 9, and $\tilde{N} = \{j \in N : \pi_1(j) = n, \cdots, n - k + 1\}$. To schedule

Algorithm 9 k-Max-Min Algorithm

```
1. If $k > 1$,
    1.1. compute the busy duration of each machine $i \in M$, given the job set $N$ as $\tau_i^{(N)} = \sum_{j \in N} p_{ij}$.
    1.2. Compute the set of candidate jobs to be the last job to complete as $I_N = \arg\min_{j \in N} \max_{i \in M}(\tau_i^{(N)} - p_{ij})$.
    1.3. For each job $j \in I_N$, run Algorithm 9 for $N \leftarrow N \setminus \{j\}$, and $k \leftarrow k - 1$ and denote the output permutation by $\pi^j$. Assign $\pi^j(j) = n$ for $j \in I_N$.
    1.4. Compare the output permutations $\{\pi^j\}$, and set $\pi_1$ to be the one whose corresponding completion time vector dominates the others in the sense of min-max fairness.
2. Else ($k = 0$), $\tilde{N} = \varnothing$, $\pi_1 = \varnothing$.
```

remaining jobs, we can compute a random permutation over remaining jobs $N \setminus \tilde{N}$, and modify it by exchanging jobs' positions according to Lemma 20 to get a permutation π_2. We can then use $\pi = [\pi_2, \pi_1]$ to schedule all jobs.

Correctness of Algorithm 9

Consider a machine i. The time that this machine needs to process all its associated tasks is given by $\tau_i^{(N)}$ as defined in line 1.1. Therefore, there exists a task (i, j) that completes at time $\tau_i^{(N)}$. Also, the completion time of the last job in any optimal schedule is equal to $\tau^{(N)} = \max_{i \in M} \tau_i^{(N)}$, which is the optimal makespan (5.2). Now the algorithm needs to decide which job should it actually complete last in the schedule. Assume that it chooses job j as the last job to complete (equivalently, $\pi(j) = n$), then the second-largest completion time across all the jobs will be equal to

$$\tau^{(N \setminus \{j\})} = \max_{i \in M} \tau_i^{(N \setminus \{j\})} = \max_{i \in M}(\tau_i^{(N)} - p_{ij}).$$

Hence, the algorithm finds the set of jobs I_N such that $\tau^{(N \setminus \{j\})}$ is minimized for $j \in I_N$. Note that this is necessary in order to achieve a min-max fair vector. Also, note that the maximization in line 1.2 of the algorithm is over the set M and not M_j, for all $j \in N$, to ensure that position n is assigned to a job with the largest completion time. Applying a similar argument, we conclude that Algorithm 9 correctly finds the last k jobs in the optimal schedule.

Time Complexity of Algorithm 9

Observe that the size of set I_N (line 1.2) is at most n. This implies that running time of the algorithm is $O(kmn^k)$ which is polynomial in input size for a fixed value of k. If we set $k = n$, we need to check all the $n!$ possible permutations to find out the optimal solution. As we can observe from execution of Algorithm 9, the reason that we need to consider all possibilities for the optimal permutation of jobs (that can blow to $n!$) is that size of candidate set I_N is generally greater than one. Hence, the Algorithm requires to check which candidate job it should choose for each position. In the case that there is a unique candidate job at each iteration, the optimal permutation can be computed in $O(mn^2 + mn\log(p))$ time, where p is the maximum task processing time.

5.5.2 Perturbation-Based Scheduling Algorithm

Algorithm 10 Perturbation-Based Algorithm

```
1. Choose a constant $\epsilon > 0$.
2. For every job $j \in N$, draw a number $\epsilon_j$ randomly from interval $[0, \epsilon]$. Then update its tasks'
   processing times $p_{ij} \leftarrow p_{ij} + \epsilon_j$.
3. For $l = n$ to 1, compute the busy duration of each machine $i \in M$ corresponding to set $N$, as in
   line 1.1 of Algorithm 9.
4. Let $I_N = \arg\min_{j \in N} \max_{i \in M}(\tau_i^{(N)} - p_{ij})$.
5. If $|I_N| \neq 1$, go to line 2. Else, set the $l$-th position in the permutation to be the unique job
   $j^\star \in I_N$, i.e., $\pi(j^\star) = l$, and update $N \leftarrow N \setminus \{j^\star\}$.
6. Schedule jobs (with the original processing times) according to the obtained permutation $\pi$.
```

Algorithm 10 gives a description of our perturbation-based algorithm to schedule multi-task jobs so as to approximate the min-max completion time vector, in the single-objective approximation sense (Section 5.4.2). At a high level, given an instance of the problem, we perturb the tasks' processing times with a small random noise. This is an attempt to ensure in execution of Algorithm 9, the number of candidate jobs calculated in line 2 reduces to 1 with high probability.

For each job j, we draw a noise ϵ_j uniformly at random from interval $[0, \epsilon]$. Define $p'_{ij} = p_{ij} + \epsilon_j$ to be the processing times in the perturbed instance. Similar to Algorithm 9, for the perturbed instance, we compute the optimal permutation starting from the last position n. The perturbation noises in practice are not real numbers, hence, the probability that the set of candidate jobs for the l-th position, $l = 1, \ldots, n$, contains more than one job is small but not zero. To resolve possible collisions in a candidate set, we have lines 5 in Algorithm 10.

Evaluation of Algorithm 10

Consider an instance of our problem. Let π denote the permutation of jobs computed by Algorithm 10. We use optimal objective value of IP (5.4) to measure the distance of the computed solution by Algorithm 10 to the optimal solution.

Theorem 10. *Let π be the permutation of jobs computed by Algorithm 10 and C denote the objective value of IP (5.4) according to this permutation. Also let OPT be the objective value of IP for an optimal min-max fair schedule. Then, $C \leq OPT + g(\epsilon)$, where $g(\epsilon)$ is a strictly decreasing function in ϵ, and $g(\epsilon) \to 0$ as $\epsilon \to 0$.*

We refer to the instance before applying perturbation as *original instance*. Recall that optimal solution of IP (5.4) is equivalent to the optimal max-min solution for the original instance; therefore, difference of the two objective values C and OPT, denoted by $g(\epsilon)$ is a sound metric to evaluate the quality of permutation π computed by Algorithm 10 for the original instance. Moreover, note that we can choose any ϵ by considering sufficiently large number of bits to represent the perturbation noise which incurs greater complexity. This issue is addressed in Subsection 5.5.2.

Proof Overview. The permutation π computed by Algorithm 10 is optimal for the perturbed instance. Therefore, by Lemma 19, π yields to the smallest objective value, $\widetilde{\text{OPT}}$, for IP (5.4) (when equipped with weights that correspond to the perturbed instance). Next, we apply the optimal permutation of the original instance, $\pi^\star$, on the perturbed instance and use $\bar{C}$ to denote its IP's value. We find the relationship between OPT and C, by comparing their values with $\widetilde{\text{OPT}}$ and

$\bar{C}$. It follows that $g(\epsilon) = (n^2+1)\epsilon \sum_{l=1}^{n} l w_l + \epsilon^2 f(\epsilon)$, for a polynomial function f. We omit the details. □

Time Complexity of Algorithm 10

Let b denote the number of bits used to represent the perturbation noises. The probability of having more than one job in set I_N in the first iteration is less than $\binom{n}{2} \times 2^{-b}$ by the union bound. Therefore, the probability of not encountering any collision in I_N is at least $1 - 2^{-(b+1)}n^2$. Choosing $b = 3 \times \log(n)$, the average number of times we should execute the algorithm to pass the first iteration successfully is less than $\frac{2n}{2n-1} \leq 2$. Applying the same argument, the average number of times needed to successfully complete all the iterations is polynomial in the input size. Therefore, Algorithm 10, on average, has polynomial time complexity in the input size of the original instance (i.e., $O(mn^2 + mn\log(p))$). In simulations for Google trace (Section 5.7), the algorithm always found each position successfully in one try.

5.6 General Utility Functions

The main obstacle in extending the results in Section 5.4 and 5.5 to unequal utility functions is that the jobs' positions in the optimal permutation, based on jobs' completion times, may not be the same as the jobs' positions according to jobs' utilities. Algorithm 9 used the fact that for any set of jobs N, there exists a job that completes at the optimal makespan $\tau^{(N)}$ (Equation (5.2)). This gives the min-max of completion times and also helps us decide which job to schedule last. However, in the case of unequal utility functions, the job that is scheduled last with the largest completion time may not be the job with the worst utility. Therefore, Algorithm 9 cannot be generalized to find the last k jobs with the worst utilities in the case of general utility functions.

Nevertheless, we present a generalization of the perturbation-based algorithm (Algorithm 10) to unequal utility functions. Since utility functions are assumed to be Lipschitz continuous (bounded first derivative), we can choose the noise parameter ϵ small enough such that job utilities do not change dramatically after perturbing task processing times. The algorithm is essentially the same as Algorithm 10, except that we do not update processing times in line 2, and instead in line 4, the set of candidate jobs is computed as

$$I_N = \arg\max_{j \in N} \min_{i \in M_j} U_j(\tau_i^{(N)} + \epsilon_j).$$

Note that the positions of jobs in the obtained permutation π by this algorithm, is neither the same as the positions based on the sorted completion time vector (Definition 11), nor the same as the positions based on the sorted utility vector. Nevertheless, we can use this permutation π to schedule jobs. We evaluate the performance of this algorithm empirically in simulations.

5.7 Simulation Results

In this section, we evaluate the performance of our algorithms using a real traffic trace from a large Google cluster [151]. The original trace is based on ~11000 servers over a month long period. In our experiments, we filter jobs and consider a set of jobs with at most 200 number of tasks which are about 99% of all the jobs in the *production* class. Also, in order to have reasonable traffic density on each machine (since otherwise the problem is trivial), we consider a cluster with 200 machines and randomly map machines of the original set to machines of this set. In simulations, we choose parameter ϵ in Algorithm 10 and its generalized version to be 10^{-4} times the smallest task processing time in the data set. For brevity, in Figures, we refer to both Algorithm 10 and its generalized version as PBA (*Perturbation-Based Algorithm*).

We evaluate the performance of our algorithms in two cases:

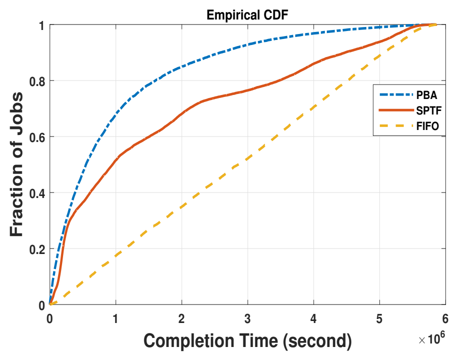
- **Equal Utility Functions:** When all the jobs have the same utility function, lexicographic max-min of utilities is equivalent to lexicographic min-max of completion times (by Lemma 17). We compare Algorithm 10 (PBA) with *First-In First-Out* (FIFO), and *Shortest Processing Time First* (SPTF). In FIFO, we list jobs based to their arrival times and schedule tasks on each machine according to this list. In SPTF, we list tasks on each machine in non-increasing order of their processing times, and schedule tasks starting from the first task in this list.

- **General Utility Functions:** We consider linear utility functions for jobs with different slopes that capture the priority information which is available for each job in the data set. In this case, we compare the performance of generalized Algorithm 10 as described in Section 5.6 (PBA), *First-In First-Out* (FIFO), and *Largest Utility First* (LUF). In LUF, we consider a utility for each task, using the utility function of its corresponding job. Then on each machine at any time, we list tasks according to their utility values, and schedule the task that gives the largest utility upon completion, then move to the next task, and so on.

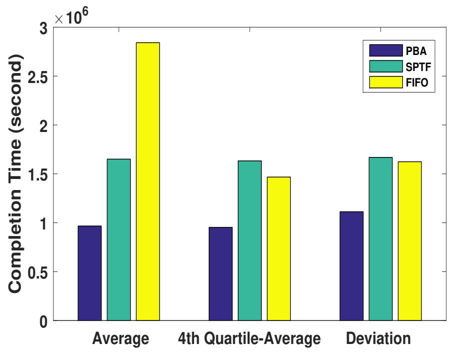
We examine algorithms by looking at *Cumulative Distribution Function* (CDF) for job completion times and utilities, in online and offline setting, with equal and unequal utility functions. In addition, we report 3 performance metrics:

- *Average*: the average of completion times of jobs (in the case of equal utility functions), or the average of their utilities (in the case of unequal utility functions).
- *4th Quartile-Average*: the average of the worst 25% of completion times or utilities among jobs. This metric indicates how much each algorithm starves long or low-utility jobs compared to the average.
- *Deviation*: the standard deviation of the job completion times (or their utilities) from the average, which is a metric of overall fairness to all jobs

In the case of equal utility functions, we report the results for job completion times, hence, smaller average and smaller 4th quartile-average are preferable. In the case of general utilities, we report the results for job utility values, hence in this case, larger average and larger 4th quartile-average are preferable. Moreover, in both cases, smaller deviation value for an algorithm shows that it has a better overall fairness.

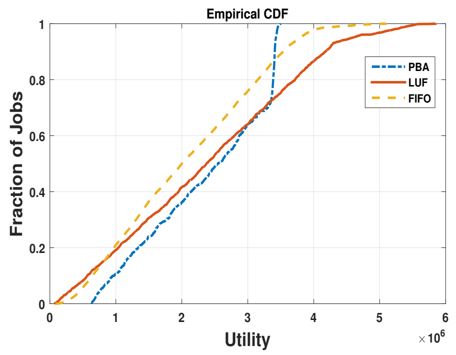


(a) Empirical CDF.

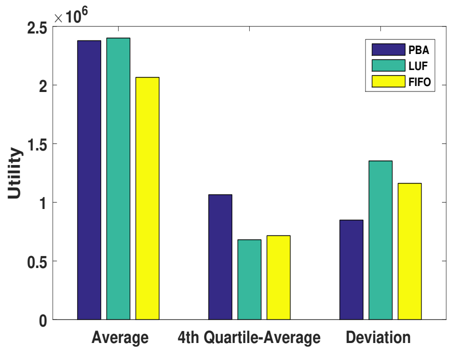


(b) Average and Deviation.

Figure 5.1: Job *completion times* under PBA, SPTF, and FIFO in the offline setting. Lower average and lower deviation is better.



(a) Empirical CDF.



(b) Average and Deviation.

Figure 5.2: Job *utilities* under PBA, LUF, and FIFO, in the offline setting. Higher averages and lower deviation is better.

5.7.1 Offline Setting

Equal Utility Functions

Figure 5.1a depicts the empirical CDF of PBA, SPTF, and FIFO. Furthermore, Figure 5.1b shows the three aforementioned performance metrics (Average, 4th Quartile-Average, and Deviation) for job completion times. Not only our algorithm is better in terms of fairness, as shown by its deviation which is 0.65 of deviation of the other algorithms, and does not starve long jobs compared to other algorithms, but interestingly it also improves the average job completion time by a factor of almost 1.7 and 3, compared to SPTF and FIFO, respectively.

General Utility Functions

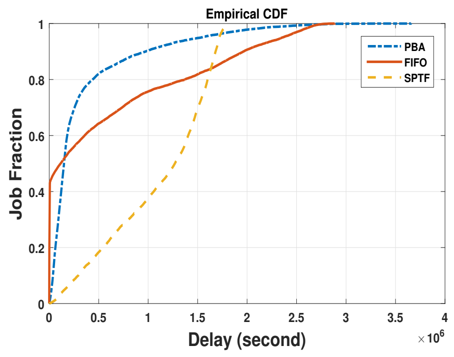
In the data set, each jobs has a priority that roughly represents how sensitive it is to latency. There are 9 different values of job priorities. For job j, we consider the utility function $U_j(t) = P_j \times (\tau - t)$, where τ is the makespan of completing all the jobs (a constant just to ensure utilities are positive) and P_j is the priority of job j.

Figure 5.2a shows the empirical CDF of PBA, LUF, and FIFO, and Figure 5.2b shows the average, 4th quartile-average, and deviation of jobs' obtained utilities. The worst utility among all the jobs under PBA is 9.5 and 6.9 times greater than the worst utility under LUF and FIFO, respectively. Note that, the CDF plot of PBA is sharper around its average value. PBA reduces deviation in obtained utilities, compared to LUF and FIFO, by a factor of 1.6 and 1.4, respectively, while it achieves almost the same average utility as LUF.

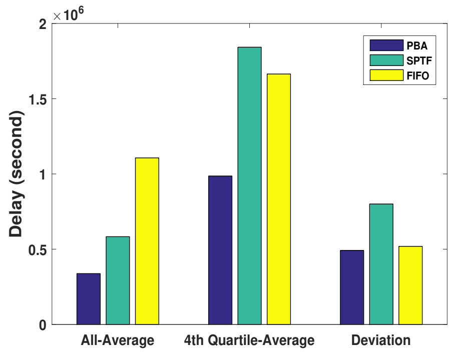
5.7.2 Online Setting

In the online setting, jobs arrive according to the arrival times information in the data set. Upon arrival of a job, SPTF updates its list and proceeds with the new list. However, it does not preempt an ongoing task in a machine. Similar to SPTF, LUF updates its list upon arrival of a job and proceeds with the new list in a non-preemptive fashion.

To extend our algorithm to online setting, we choose a parameter δ that is tunable. We divide time into time intervals of length δ time-units. At the beginning of each interval, we run the offline algorithm on the set of jobs consisting of jobs that are not scheduled yet and those that arrived in the previous interval. Further, tasks on the boundary of intervals are processed non-preemptively, i.e., if some task is running in some machine according to the previously computed schedule, we let it finish and then proceed with the new schedule. It is preferred to start with a smaller value of δ at the beginning, to avoid delaying the initial jobs in the system for δ amount of time before starting scheduling them. Therefore, we use an adaptive choice of δ to improve the performance of our online algorithm. We choose the length of the i-th interval, δ_i, as

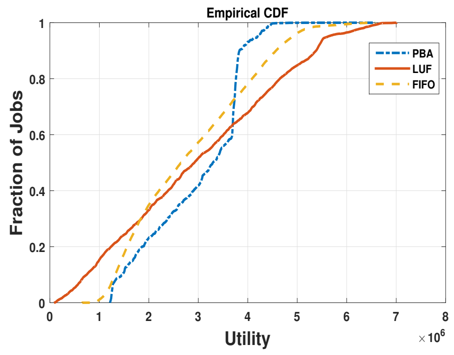


(a) Empirical CDF.

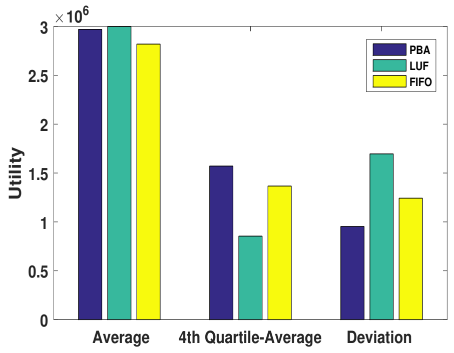


(b) Average and Deviation.

Figure 5.3: Job *delays* under PBA, SPTF, and FIFO, in the online setting. Lower averages and lower deviation is better.



(a) Empirical CDF.



(b) Average and Deviation.

Figure 5.4: Job *utilities* under PBA, LUF, and FIFO, in the online setting. Higher averages and lower deviation is better.

$$\delta_i = \delta_0/(1 + \alpha \times \exp(-\beta i)), \; i = 1, 2, \cdots$$

We choose $\delta_0 = 3.3 \times 10^5$ seconds, and $\alpha = 50$ and $\beta = 3$. All the jobs arrive over a time horizon of 3.3×10^6 seconds.

Equal Utility Functions

Figure 5.3a and 5.3b show the performance of PBA, SPTF, and FIFO in the online setting. We present the results in terms of job delay, which is the time between a job arrival and its completion time. PBA improves the average job delay by a factor of 1.7 and 3.3, compared to SPTF and FIFO. It also achieves better fairness by a factor 1.9 and 1.7 compared to SPTF and FIFO for the 4th quartile-average.

General Utility Functions

In the online setting, variable t used in the job utility function is measured from arrival of job j to the system. Figure 5.4a shows the empirical CDF of PBA, SPTF, and FIFO. Further, Figure 5.4b shows the average and deviation of jobs' obtained utilities. The smallest utility value among all the jobs under PBA is 1.9 and 14.6 times greater than the smallest utility value of jobs under FIFO and LUF, respectively. PBA also improves utility deviation compared to LUF and FIFO by a factor of 1.8 and 1.3, respectively.

Chapter 6: Scheduling Parallel-Task Jobs Subject to Packing and Placement Constraints

6.1 Introduction

Modern parallel computing frameworks, e.g. Hadoop and Spark [11, 12], have enabled large-scale data processing in computing clusters. In such frameworks, the data is typically distributed across a cluster of machines and is processed in multiple stages. In each stage, a set of tasks are executed on the machines, and once all the tasks in the stage finish their processing, the job is finished or moved to the next stage. For example, in MapReduce [4], in the map stage, each map task performs local computation on a data block in a machine and writes the intermediate data to the disk. In the reduce stage, each reduce task pulls intermediate data from different maps, merges them, and computes its output. While the reduce tasks can start pulling data as map tasks finish, the actual computation by the reduce tasks can only start once all the map tasks are done and their data pieces are received. Further, the job is not completed unless all the reduce tasks finish. Similarly, in Spark [152], the computation is done in multiple stages. The tasks in a stage can run in parallel, however, the next stage cannot start unless the tasks in the preceding stage(s) are all completed.

We refer to such constraints as *synchronization* constraints, i.e., a stage is considered completed only when all its tasks finish their processing. Such synchronizations could have a significant impact on the jobs' latency in parallel computing clusters [153, 154, 155, 156, 152]. Intuitively, an efficient scheduler should complete all the (inhomogeneous) tasks of a stage more or less around the same time, while prioritizing the stages of different jobs in an order that minimizes the overall latency in the system. Note that the scheduler can only make scheduling decisions for the stages that have been released from various jobs up to that point (i.e., those that their preceding stages have been completed). In our model, we use the terms stage and job interchangeability.

Another main feature of parallel computing clusters is that jobs can have diverse tasks and processing requirements. This has been further amplified by the increasing complexity of workloads, i.e., from traditional batch jobs, to queries, graph processing, streaming, and machine learning jobs, that all need to share the same cluster. The cluster manager (*scheduler*) serves the tasks of various jobs by reserving their requested resources (e.g. CPU, memory, etc.). For example, in Hadoop [11], the resource manager reserves the tasks' resource requirements by launching "*containers*" in machines. Each container reserves required resources for processing of a task. To improve the overall latency, we therefore need a scheduler that *packs* as many tasks as possible in the machines, while retaining their resource requirements.

In practice, there are further placement constraints for processing tasks on machines. For example, each task is preferred to be scheduled on one of the machines that has its required data block [4, 13] (a.k.a. data locality), otherwise processing can slow down due to data transfer. The data block might be stored in multiple machines for robustness and failure considerations. However, if all these machines are highly loaded, the scheduler might actually need to schedule the task in a less loaded machine that does not contain the data.

Despite the vast scheduling literature, scheduling algorithms with theoretical results (approximation ratios) are mainly based on simple models, where each machine processes one task at a time, each job is a single task, or tasks can be processed on any machine arbitrarily (see Related Work in Section 6.1.1). Such models *do not* fully capture the *modern features* of data-parallel computing clusters, namely,

- *packing*: each machine is capable of processing multiple tasks at a time subject to its capacity.
- *synchronization*: tasks that belong to the same job have a collective completion time which is determined by the slowest task in the collection.
- *placement constraint*: a task's processing time is machine-dependent and a task is typically preferred to be processed on a subset of machines (e.g. where its input data block is located).

Further, each task at each time can get processed on at most a single machine.

The goal of this work is to design scheduling algorithms, with theoretical guarantees, under the above features of modern parallel computing clusters. For simplicity, we consider one dimension for task resource requirement (e.g. memory). While task resource requirements are in general multi-dimensional (CPU, memory, etc.), it has been observed that memory is typically the bottleneck resource [90, 157].

Our objective is to minimize the weighted sum of completion times of existing jobs in the system, where weights can encode different priorities for the jobs. Clearly, minimization of the average completion time is a special case of this problem with equal weights. We consider both preemptive and non-preemptive scheduling. In a *non-preemptive* schedule, a task cannot be preempted (and hence cannot be migrated among machines) once it starts processing on a machine until it is completed. In a *preemptive* schedule, a task may be preempted and resumed later in the schedule, and we further consider two cases depending on whether migration of a task among machines is allowed or not.

6.1.1 Related Work

Default cluster schedulers in Hadoop [11, 136] focus primarily on fairness and data locality. Such schedulers can make poor scheduling decisions by not packing tasks well together, or having a task running long without enough parallelism with other tasks in the same job. Several cluster schedulers have been proposed to improve job completion times, e.g. [99, 158, 159, 160, 145, 161, 162, 163, 164]. However, they either do not consider all aspects of packing, synchronization, and data locality, or use heuristics which are not necessarily efficient.

We highlight four relevant papers [145, 161, 163, 164] here. Tetris [145] is a scheduler that assigns scores to tasks based on Best-Fit bin packing and Shortest-Remaining-Time-First (SRPF) heuristic, and gives priority to tasks with higher scores. The data locality is encoded in scores by imposing a remote penalty to penalize use of remote resources. Borg [161] packs multiple tasks of jobs in machines from high to low priority, modulated by a round-robin scheme within a priority to ensure fairness across jobs. The scheduler considers data locality by assigning tasks to machines that already have the necessary data stored. The papers by [163] and [164] focus on single-task jobs and study the mean delay of tasks under a stochastic model where if a task is scheduled on one of the remote servers that do not have the input data, its average processing time will be larger, by a multiplicative factor, compared to the case that it is processed on a local server that contains the data. They propose algorithms based on Join-the-Shortest-Queue and Max-Weight (JSQ-MW) to incorporate data locality in load balancing. This model is generalized by [164] to more levels of data locality. However, these models do not consider any task packing in servers or synchronization issue among multiple tasks of the same job.

From a theoretical perspective, our problem of scheduling parallel-task jobs with synchronization, packing, and placement constraints, can be seen as a generalization of the concurrent open shop (COS) problem [138]. Unlike COS, where each machine processes one task at a time and each task can be processed on a specific machine, in our model a machine can process (pack) multiple tasks simultaneously subject to its capacity, and there are further task placement constraints for assigning tasks to machines. Minimizing the weighted sum of completion times in COS, is known to be APX-hard [139], with several 2-approximation algorithms [165, 139, 140, 75, 82, 166]. There is also a line of research on the parallel tasks scheduling (PTS) problem [167]. In PTS, each job is only a single task that requires a certain amount of resource for its processing time, and can be served by any machine subject to its capacity. This differs from our model where each job has multiple tasks, each task can be served by a set of machines, and the job's completion time is determined by its last task. Minimizing the weighted sum of completion times in the PTS is also NP-complete in the strong sense [168]. In the case of a single machine, [169] proposed a non-preemptive algorithm that can achieve approximation ratio of 7.11, and a preemptive algorithm, called *PSRS*, that can achieve approximation ratio of 2.37. In the case of multiple machines, there is only one result in the literature which is a 14.85-approximation non-preemptive algorithm [170].

We emphasize that our setting of parallel-task jobs, subject to synchronization, packing, and placement constraints, is significantly more challenging than the COS and PTS problems, and algorithms from these problems *cannot* be applied to our setting. To the best of our knowledge, this is the first work that provides constant-approximation algorithms for this problem subject to synchronization, packing, and placement constraints,

6.1.2 Main Contributions

We briefly summarize our main results and describe our techniques below. We propose scheduling algorithms for three cases:

- **Task Migration Allowed.** When migration is allowed, a task might be preempted several times and resume possibly on a different machine within its placement-feasible set. Our algorithm in this case is based on greedy scheduling of task fractions (fraction of processing time of each task) on each machine, subject to capacity and placement constraints. The task fractions are found by solving a relaxed linear program (LP), which divides the time horizon into geometrically-increasing time intervals, and uses *interval-indexed variables* to indicate what fraction of each task is served at which interval on each machine. We show that our scheduling algorithm has an approximation ratio better than $(6 + \epsilon)$, for any $\epsilon > 0$.

- **Task Migration Not Allowed.** When migration is not allowed, the schedule can be non-preemptive, or preemptive while all preemptions occur on the same machine. In this case, our algorithm is based on mapping tasks to proper time intervals on the machines. We utilize the interval-indexed variables to form a relaxed LP. We then utilize the LP's optimal solution to construct a *weighted bipartite graph* representing tasks on one side and machine-intervals on the other side, and fractions of tasks completed in machine-intervals as weighted edges. We then use an integral matching in this graph to construct a mapping of tasks to machine-intervals. Finally, the tasks mapped to intervals of the same machine are packed in order and non-preemptively by using a greedy policy. We prove that this non-preemptive algorithm has an approximation ratio better than 24. Further, we show that the algorithm's solution is also a 24-approximation for the case that preemption on the same machine is allowed.

- **Preemption and Single-Machine Placement Set.** When preemption is allowed, and there is a specific machine for each task, we propose an algorithm with an improved approximation ratio of 4. The algorithm first finds a proper ordering of jobs, by solving a relaxed LP of our scheduling problem. Then, for each machine, it lists its tasks, with respect to the obtained ordering of jobs, and apply a simple greedy policy to pack tasks in the machine subject to its capacity. The methods of LP relaxation and list scheduling have been used in scheduling literature; however, the application and analysis of such techniques in presence of packing, placement, and synchronization is very different.

- **Empirical Evaluations.** We evaluate the performance of our preemptive and non-preemptive algorithms compared with the prior approaches using a Google traffic trace [151]. We also present online versions of our algorithms that are suitable for handling dynamic job arrivals. Our $4-$approximation preemptive algorithm outperforms PSRS [169] and Tetris [145] by up to 69% and 79%, respectively, when jobs' weights are determined using their priority information in the data set. Further, our non-preemptive algorithm outperforms JSQ-MW [163] and Tetris [145] by up to 81% and 175%, respectively, under the same placement constraints. Note that, since these algorithms do not consider all aspects of packing, synchronization, and data locality, we combined them with reasonable heuristics to enforce all the constraints in our settings.

This chapter is based on the results of the paper [171].

6.2 Formal Problem Statement

Cluster and Job Model. Consider a collection of machines $\mathcal{M} = \{1, ..., M\}$, where machine i has capacity $m_i > 0$ on its available resource. We use $\mathcal{J} = \{1, ..., N\}$ to denote the set of existing jobs (stages) in the system that need to be served by the machines. Each job $j \in \mathcal{J}$ consists of a set of tasks $\mathcal{K}_j$, where we use (k, j) to denote task k of job j, $k \in \mathcal{K}_j$. Task (k, j) requires a specific amount a_{kj} of resource for the duration of its processing. Machine i can process multiple tasks at the same time, however, the sum of resource requirements of tasks running in machine i should not exceed its capacity m_i at any time.

Task Processing and Placement Constraint. Each task (k, j) can be processed on a machine from a specific set of machines $\mathcal{M}_{kj} \subseteq \mathcal{M}$. We refer to $\mathcal{M}_{kj}$ as the *placement set* of task (k, j). For generality, we let p^i_{kj} denote the processing time of task (k, j) on machine $i \in \mathcal{M}_{kj}$. Such placement constraints can model data locality. For example, we can set $\mathcal{M}_{kj}$ to be the set of machines that have task (k, j)'s data, and $p^i_{kj} = p_{kj}$, $i \in \mathcal{M}_{kj}$. Or, we can consider $\mathcal{M}_{kj}$ to be as large as $\mathcal{M}$, and incorporate the data transfer cost as a penalty in the processing time on machines that do not have the task's data.

Throughout the chapter, we refer to a_{kj} as size or resource requirement of task (k, j), and to p^i_{kj} as its length, duration, or processing time on machine i. We also define the volume of task (k, j) on machine i as $v^i_{kj} = a_{kj} p^i_{kj}$. Without loss of generality, we assume processing times are nonnegative integers and duration of the smallest task is at least one. This can be done by defining a proper time unit (slot) and representing the task durations using integer multiples of this unit.

Synchronization Constraint. Tasks can be processed in parallel on their corresponding machines; however, a job is considered completed only when all of its tasks finish. Hence, using C_{kj} to denote the completion time of task (k, j), the completion time of job j, denoted by C_j, satisfies

$$C_j = \max_{k \in \mathcal{K}_j} C_{kj}. \tag{6.1}$$

Let $\mathbb{1}(i \in \mathcal{M}_{kj})$ be the indicator function which is 1 if $i \in \mathcal{M}_{kj}$, and 0 otherwise. Define

$$T = \max_{i \in \mathcal{M}} \sum_{j \in \mathcal{J}} \sum_{k \in \mathcal{K}_j} p^i_{kj} \mathbb{1}(i \in \mathcal{M}_{kj}), \tag{6.2}$$

which is clearly an upper bound on the time required for processing all the jobs. We define 0-1 variables $X^i_{kj}(t)$, $i \in \mathcal{M}$, $j \in \mathcal{J}$, $k \in \mathcal{K}_j$, $t \leq T$, where $X^i_{kj}(t) = 1$ if task (k, j) is served at time slot t on machine i, and 0 otherwise. We also make the following definition.

Definition 13 (Height of Machine i at time t)**.** *The height of machine i at time t, denoted by $h_i(t)$,* *is the sum of resource requirements of the tasks running at time t in machine i, i.e.,*

$$h_i(t) = \sum_{j \in \mathcal{J}, k \in \mathcal{K}_j} a_{kj} X^i_{kj}(t). \tag{6.3}$$

Given these definitions, a valid schedule $X^i_{kj}(t) \in \{0, 1\}$, $i \in \mathcal{M}$, $j \in \mathcal{J}$, $k \in \mathcal{K}_j$, $0 < t \leq T$, must satisfy the following three constraints:

(i) *Packing*: the sum of resource requirements of the tasks running in machine i at time t (i.e., tasks with $X^i_{kj}(t) = 1$) should not exceed machine i's capacity, i.e., $h_i(t) \leq m_i$, $\forall t \leq T$, $\forall i \in \mathcal{M}$.

(ii) *Placement*: each task at each time can get processed on at most a single machine selected from its feasible placement set, i.e., $\sum_{i \in \mathcal{M}_{kj}} X^i_{kj}(t) \leq 1$, and $X^i_{kj}(t) = 0$ if $i \notin \mathcal{M}_{kj}$.

(iii) *Processing*: each task must be processed completely. Noting that $X^i_{kj}(t)/p^i_{kj}$ is the fraction of task (k, j) completed on machine i in time slot t, we need $\sum_{i \in \mathcal{M}_{kj}} \sum_{t=1}^{T} X^i_{kj}(t)/p^i_{kj} = 1$.

Preemption and Migration. We consider three classes of scheduling policies. In a non-preemptive policy, a task cannot be preempted (and hence cannot be migrated among machines) once it starts processing on its corresponding machine until it is completed. In a preemptive policy, a task may be preempted and resumed several times in the schedule, and we can further consider two subcases depending on whether migration of a task among machines is allowed or not. Note that when migration is not allowed, the scheduler must assign each task (k, j) to one machine $i \in \mathcal{M}_{kj}$ on which the task is (preemptively or non-preemptively) processed until completion.

Main Objective. Given positive weights w_j, $j \in \mathcal{J}$, our goal is to find valid non-preemptive and preemptive (under with and without migrations) schedules of jobs (their tasks) in machines, so as to minimize the sum of weighted completion times of jobs, i.e.,

$$\text{minimize} \sum_{j \in \mathcal{J}} w_j C_j. \tag{6.4}$$

The weights can capture different priorities for jobs. Clearly the case of equal weights reduces the problem to minimization of the average completion time.

Here, we use the 3-field notation to specify our problems. While we utilize some of the notations from the scheduling literature, we need to define new ones to capture all the constraints in our model. We consider the following problems:

- PRP|mgr|$\sum_j w_j C_j$
- PRP| |$\sum_j w_j C_j$ and PRP|pmtn|$\sum_j w_j C_j$
- PDP|pmtn|$\sum_j w_j C_j$

In the first field of the notations, the first letter *P* stands for "parallel" and specifies the fact that the machines can process different tasks of a given job in parallel. The letter *R* means that the machines are "unrelated", i.e., a task has different processing times on different machines. The letter *D* stands for "dedicated" and shows that there is a dedicated machine for processing of each task. Finally, the last letter *P* stands for "packing" and shows that a machine can pack tasks subject to its capacity. In the second field, *pmts* and *mgr* indicate that processing of a task can be preempted and a task can migrated among machines, respectively. Finally, the objective function is specified in the third field.

6.3 Scheduling When Migration is Allowed

We first consider the case that migration of tasks among machines is allowed. This is equivalent to PRP|mgr|$\sum_j w_j C_j$. In this case, we propose a preemptive algorithm, called SynchPack-1, with approximation ratio $(6+\epsilon)$ for any $\epsilon > 0$. We will use the construction ideas and analysis arguments for this algorithm to construct our preemptive and non-preemptive algorithms when migration is prohibited in Section 6.4.

In order to describe SynchPack-1, we first present a relaxed linear program. We will utilize the optimal solution to this LP to schedule tasks in a preemptive fashion.

6.3.1 Relaxed Linear Program (LP1)

Recall that without loss of generality, the processing times of tasks are assumed to be integers (multiples of a time unit) and therefore $C_j \geq p^i_{kj} \geq 1$ for all $j \in \mathcal{J}$, $k \in \mathcal{K}_j$, and $i \in \mathcal{M}_{kj}$. We use interval indexed variables using geometrically increasing intervals (see, e.g., [172, 173, 69]) to formulate a linear program for our problem.

Let $\epsilon > 0$ be a constant. We choose L to be the smallest integer such that $(1+\epsilon)^L \geq T$ (recall T in (6.2)). Subsequently define

$$d_l = (1+\epsilon)^l, \text{ for } l = 0, 1, \cdots, L, \tag{6.5}$$

and define $d_{-1} = 0$. We partition the time horizon into time intervals $(d_{l-1}, d_l]$, $l = 0, ..., L$. Note that the length of the l-th interval, denoted by Δ_l, is

$$\Delta_0 = 1, \;\; \Delta_l = \epsilon(1+\epsilon)^{l-1} \;\; \forall l \geq 1. \tag{6.6}$$

We define z^{il}_{kj} to be the fraction of task (k, j) (*fraction of its required processing time*) that is processed in interval l on machine $i \in \mathcal{M}_{kj}$.

To measure completion time of job j, for each interval l, we define an integer variables x_{jl} which is 1 if job j finishes in interval l and 0 otherwise. Consider the following constraints, $\forall j \in \mathcal{J}$:

$$\sum_{l'=0}^{l} x_{jl'} \leq \sum_{l'=0}^{l} \sum_{i \in \mathcal{M}_{kj}} z^{il'}_{kj}, \; k \in \mathcal{K}_j, \; l = 0, \ldots, L \tag{6.7a}$$

$$\sum_{l=0}^{L} x_{jl} = 1, \;\; x_{jl} \in \{0, 1\}, \;\; l = 0, \cdots, L. \tag{6.7b}$$

Note that (6.7b) implies that only one of the variables $\{x_{jl}\}_{l=0}^{L}$ can be nonzero (equal to 1). (6.7a) implies that x_{jl} can be 1 only for one of the intervals $l \geq l^\star$ where $l^\star$ is the interval in which the last task of job j finishes its processing. Now define,

$$C_j = \sum_{l=0}^{L} d_{l-1} x_{jl} \quad j \in \mathcal{J}. \tag{6.8}$$

If we can guarantee that $x_{jl^\star} = 1$ for $l^\star$ as defined above, then C_j will be equal to the starting point $d_{l^\star-1}$ of that interval, and the actual completion time of job j will be bounded above by $d_{l^\star} = (1+\epsilon)C_j$, thus implying that C_j is a reasonable approximation for the actual completion time of job j. This can be done by minimizing the objective function in the following linear program:

$$\min \sum_{j \in \mathcal{J}} w_j C_j \qquad (\mathbf{LP1}) \tag{6.9a}$$

$$\sum_{l=0}^{L} \sum_{i \in \mathcal{M}_{kj}} z_{kj}^{il} = 1, \quad k \in \mathcal{K}_j,\ j \in \mathcal{J} \tag{6.9b}$$

$$\sum_{l'=0}^{l} \sum_{i \in \mathcal{M}_{kj}} z_{kj}^{il'} p_{kj}^{i} \leq d_l, \quad k \in \mathcal{K}_j,\ j \in \mathcal{J},\ l = 0, \ldots, L \tag{6.9c}$$

$$\sum_{l'=0}^{l} \sum_{(k,j): i \in \mathcal{M}_{kj}} z_{kj}^{il'} p_{kj}^{i} a_{kj} \leq m_i d_l, \quad i \in \mathcal{M},\ l = 0, \ldots, L \tag{6.9d}$$

$$z_{kj}^{il} \geq 0, \quad k \in \mathcal{K}_j,\ j \in \mathcal{J},\ i \in \mathcal{M}_{kj},\ l = 0, \ldots, L \tag{6.9e}$$

$$\sum_{l'=0}^{l} x_{jl'} \leq \sum_{l'=0}^{l} \sum_{i \in \mathcal{M}_{kj}} z_{kj}^{il'}, \quad k \in \mathcal{K}_j,\ j \in \mathcal{J},\ l = 0, \ldots, L \tag{6.9f}$$

$$C_j = \sum_{l=0}^{L} d_{l-1} x_{jl}, \quad j \in \mathcal{J} \tag{6.9g}$$

$$\sum_{l=0}^{L} x_{jl} = 1, \quad x_{jl} \geq 0, \quad l = 0, \ldots, L,\ j \in \mathcal{J} \tag{6.9h}$$

Constraint (6.9b) means that each task must be processed completely. (6.9c) is because during the first l intervals, a task cannot be processed for more than d_l, the end point of interval l, which itself is due to requirement (ii) of Section 6.2. (6.9d) bounds the total volume of the tasks processed by any machine i in the first l intervals by $d_l \times m_i$. (6.9e) indicates that z variables have to be nonnegative.

Constraints (6.9f), (6.9h), (6.9g) are the relaxed version of (6.7a), (6.7b), (6.8), respectively, where the integral constraint in (6.7b) has been relaxed to (6.9h). To give more insight, note that (6.9f) has the interpretation of keeping track of the fraction of the job processed by the end of each time interval, which is bounded from above by the fraction of any of its tasks processed by the end of that time interval. We should finish processing of all jobs as indicated by (6.9h). Also (6.9g) computes a relaxation of the job completion time C_j, as a convex combination of the intervals' left points, with coefficients x_{jl}.

6.3.2 Scheduling Algorithm: SynchPack-1

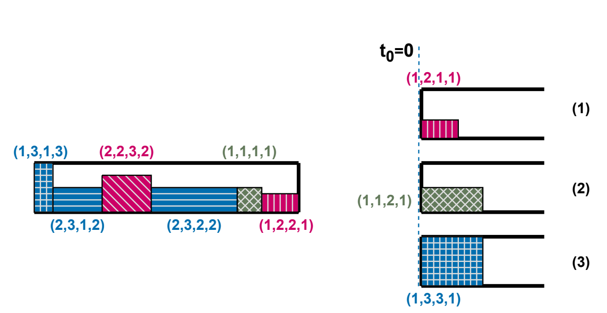
In the following, a *task fraction* (k, j, i, l) of task (k, j) corresponding to interval l, is a task with size a_{kj} and duration $z^{il}_{kj} p^{i}_{kj}$ that needs to be processed on machine i.

The SynchPack-1 (*Synchronized Packing*-1) algorithm has three main steps:

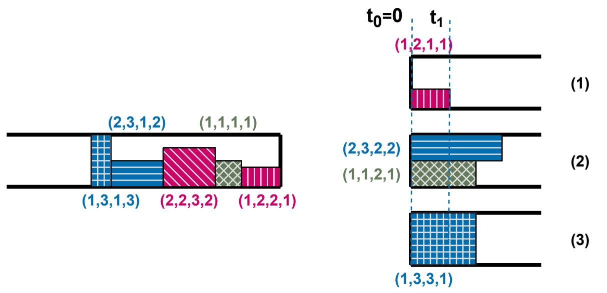
Step 1: Solve (LP1). We first solve (LP1) and obtain the optimal solution of $\{z^{il}_{kj}\}$ which we denote by $\{\tilde{z}^{il}_{kj}\}$.

Step 2: Pack task fractions greedily to construct schedule $\mathcal{S}$. To schedule task fractions, we use a greedy list scheduling policy as follows:

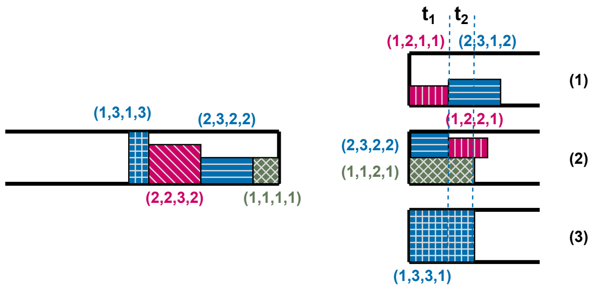
Consider an ordered list of the task fractions such that task fractions corresponding to interval l appear before the task fractions corresponding to interval l', if $l < l'$. Task fractions within each interval and corresponding to different machines are ordered arbitrarily. Let t denote a time at which the algorithm makes some scheduling decision. The algorithm scans the list starting from the first task fraction, and schedules task fraction (k, j, i, l) on machine i, if some fraction of task (k, j) is not already scheduled on some other machine at time t, and machine i has sufficient capacity, i.e., $h_i(t) + a_{kj} \leq m_i$ (recall $h_i(t)$ in Definition 13). It then moves to the next task fraction in the list, repeats the same procedure, and so on. Upon completion of a task fraction, it preempts the task fractions corresponding to higher indexed intervals on all the machines if there is some unscheduled task fraction of a lower-indexed interval in the list. It then removes the completed task fraction(s) from the list, updates the remaining processing times of the task fractions in the



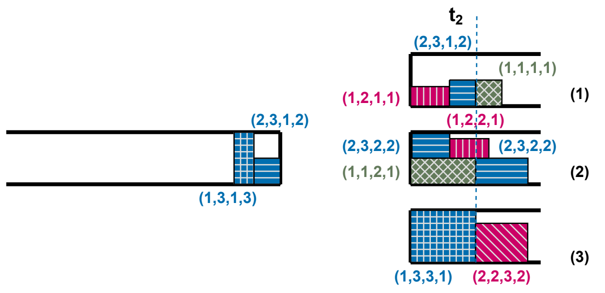
(a) A list of task fractions is given. The first three task fractions in the list are already scheduled on the machines at time $t_0 = 0$.



(b) Due to placement constraint, task fractions (1, 2, 2, 1) and (1, 1, 1, 1) cannot get scheduled. However, machine 2 can accommodate task fraction (2, 3, 2, 2).



(c) At t_1 task fraction (1, 2, 1, 1) completes and task fraction (2, 3, 2, 2) is preempted. Task fractions (1, 2, 2, 1) and (2, 3, 1, 2) are scheduled.



(d) Both task fractions (1, 1, 2, 1) and (1, 3, 3, 1) complete, and task fraction (2, 3, 1, 2) is preempted at time t_2. Then task fractions (1, 1, 1, 1), (2, 3, 2, 2), and (2, 2, 3, 2) are scheduled.

Figure 6.1: An example for execution of Step 2 of SynchPack-1 for 3 jobs in a system with 3 machines. Different tasks of a job have the same color and different patterns. Note that task fraction (1, 2, 2, 1), which is at the head of the list in Figure 6.1a, cannot get scheduled on machine 2 as task fraction (1, 2, 1, 1) (of the same task (1, 2)) is already scheduled on machine 1. At time t_1, task fraction (1, 2, 1, 1) is finished processing as shown in Figure 6.1b. At this time, while task fraction (2, 3, 2, 2) is running on machine 2 (whose corresponding interval is 2), two task fractions, namely (1, 2, 2, 1) and (1, 1, 1, 1) (whose corresponding intervals are 1), have remained unscheduled in the list. Therefore, task fraction (2, 3, 2, 2) is preempted and its remaining duration is updated. Then, the algorithm scans the list and schedules the task fractions as shown in Figure 6.1c. The next time that a completion occurs is denoted by t_2. Figure 6.1d shows the schedule at this time. The rest of the schedule can be determined in a similar fashion.

list, and starts scheduling the updated list. Note that the set of times at which scheduling decisions are made consists of time 0 and task fractions' completion times. This greedy list scheduling algorithm schedules task fractions in a preemptive fashion. We refer to the constructed schedule as $\mathcal{S}$. As an illustration, Figure 6.1 shows execution of Step 2 in a system with 3 machines and 3 jobs.

Step 3: Apply Slow-Motion technique to construct schedule $\bar{\mathcal{S}}$. Unfortunately, we cannot bound the value of objective function (6.9a) for schedule $\mathcal{S}$, since completion times of some jobs in $\mathcal{S}$ can be very long compared to the completion times returned by (LP1)[1].

Therefore, we construct a new feasible schedule $\bar{\mathcal{S}}$, by stretching $\mathcal{S}$, for which we can bound the value of its objective function. This method is referred to as *Slow-Motion* technique [174]. Let $\tilde{Z}_{kj}^i = \sum_{l=0}^{L} \tilde{z}_{kj}^{il}$ denote the total fraction of task (k, j) that is scheduled in machine i according to the optimal solution to (LP1). We refer to $\tilde{Z}_{kj}^i$ as the *total task fraction* of task (k, j) on machine i. The Slow-Motion technique works by choosing a parameter $\lambda \in (0, 1]$ randomly drawn according to the probability density function $f(\lambda) = 2\lambda$. It then stretches schedule $\mathcal{S}$ by a factor $1/\lambda$. If a task is scheduled in $\mathcal{S}$ during an interval $[\tau_1, \tau_2)$, the same task is scheduled in $\bar{\mathcal{S}}$ during $[\tau_1/\lambda, \tau_2/\lambda)$ and *the machine is left idle if it has already processed its total task fraction $\tilde{Z}_{kj}^i$ completely*. We may also shift back future tasks' schedules as far as the machine capacity allows and placement constraint is respected. Figure 6.2 shows the execution of this step on the example of Figure 6.1 for $\lambda = 1/2$.

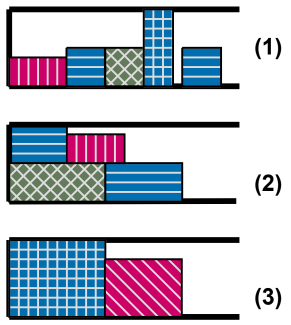
A pseudocode for SynchPack-1 can be found in Section 6.14. The obtained algorithm is a randomized algorithm; however, we will show in Section 6.10 how we can de-randomize it to get a deterministic algorithm.

6.3.3 Performance Guarantee

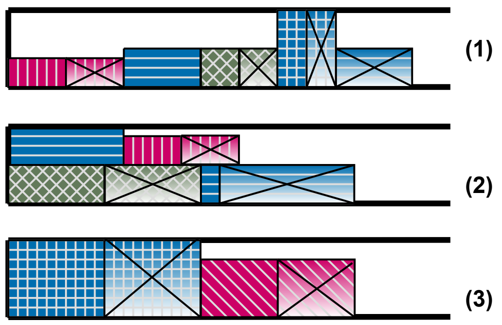
We now analyze the performance of SynchPack-1. The result is stated by the following proposition.

Theorem 11. *For any $\epsilon > 0$, the sum of weighted completion times of jobs, for the problem of parallel-task job scheduling with packing and placement constraints, under SynchPack-1, is at*

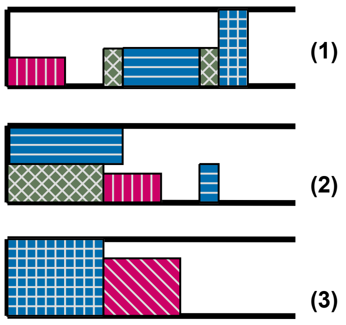
[1]This is because of how C_j is defined as a convex combination of the interval left points in constraint (6.9g). More specifically, assume job j consists of one task and completes at interval l_j, however, only a very small fraction of its task is scheduled in l_j, i.e., x_{jl_j} is very small. Furthermore, assume the rest of the task is scheduled at some interval l where $l << l_j$. Then, we can choose x_{jl_j} such that $C_j \sim d_l$ (according to (6.9g)), while the actual completion time of job j in schedule $\mathcal{S}$ can be $\sim d_{l_j}$.



(a) Schedule $\mathcal{S}$ for the example of Figure 6.1.



(b) Schedule $\bar{\mathcal{S}}$ for stretch factor $\lambda = 1/2$. The machines are left idle in the shadowed crossed parts.



(c) Final schedule after shifting back future tasks' schedules while respecting the constraints.

Figure 6.2: An example for execution of Slow-Motion technique in Step 3 of SynchPack-1. In Figure 6.2a, the final schedule of the example in Figure 6.1 is shown. Figure 6.2b shows the result after applying Slow-Motion with $\lambda = 1/2$. If a machine has already processed total task fraction of a task completely, it is left idle. For instance, consider the blue task fraction on machine 2, i.e. $(2, 3, 2, 2)$. Some portion of its schedule in the second part is shadowed and crossed and machine 2 is left idle, since machine 2 has already processed this task fraction for the total time that it does originally in Figure 6.2a. Figure 6.2c shows the result after shifting back future tasks' schedules while respecting the constraints. For instance, see the red task fraction (i.e., $(1, 2, 2, 1)$ on machine 2 and part of the green task fraction (i.e., $(1, 1, 1, 1)$) on machine 1. The idle times on the machines are left blank in Figure 6.2c. Note that this last action (shifting back future tasks' schedules) is optional.

most $(6 + \epsilon) \times OPT$.

The rest of the section is devoted to the proof of Theorem 11. We use $\tilde{C}_j$ to denote the optimal solution to (LP1) for completion time of job $j \in \mathcal{J}$. The optimal objective value of (LP1) is a lower bound on the optimal value of our scheduling problem as stated in the following lemma whose proof is provided in Section 6.9.1.

Lemma 21. $\sum_{j=1}^{N} w_j \tilde{C}_j \leq \sum_{j=1}^{N} w_j C_j^{\star} = OPT$.

Note that Constraint (6.9d) bounds the volume of all the task fractions corresponding to the first l intervals on machine i by $d_l \times m_i$. However, the (LP1)'s solution does not directly provide a feasible schedule as task fractions of the same task on different machines might overlap during the same interval, and machines' capacity constraints might be also violated as Constraint (6.9d) in (LP1) bounds the total volume of the processed tasks and ignores their sizes and durations. Next, we show under the greedy list scheduling policy (Step 2 in SynchPack-1), the completion time of task fraction (k, j, i, l) is bounded from above by $3 \times d_l$, i.e., we need a factor 3 to guarantee a feasible schedule.

Lemma 22. *Let τ_l denote the time that all the task fractions (k, j, i, l'), for $l' \leq l$, are completed in schedule $\mathcal{S}$. Then, $\tau_l \leq 3d_l$.*

Proof. Proof. Consider the non-zero task fractions (k, j, i, l'), $i \in \mathcal{M}$, $l' \leq l$ (according to an optimal solution to (LP1)). Without loss of generality, we normalize the processing times of task fractions to be positive integers, by defining a proper time unit and representing the task durations using integer multiples of this unit. Let D_l and T_l be the value of d_l and τ_l using the new unit. Let $i^\star$ denote the machine that schedules the last task fraction among the non-zero task fractions of the first l intervals. Note that T_l is the time that this task fraction completes. If $T_l \leq D_l$, then $T_l \leq 3D_l$ and the lemma is proved. Hence consider the case that $T_l > D_l$.

Define $h_{il}(t)$ to be the height of machine i at time t in schedule $\mathcal{S}$ considering only the task fractions of the first l intervals. First we note that,

$$\sum_{l'=0}^{l} \sum_{(k,j):i\in\mathcal{M}_{kj}} z_{kj}^{il'} p_{kj}^{i} a_{kj} \overset{(a)}{=} \sum_{t=1}^{T_l} h_{il}(t) \overset{(b)}{\leq} m_i D_l, \quad \forall i \in \mathcal{M} \tag{6.10}$$

Using the definition of $h_{il}(t)$, the right-hand side of Equality (a) is the total volume of task fractions corresponding to the first l intervals that are processed during the interval $(0, T_l]$ on machine i, which is the left-hand side. Further, Inequality (b) is by Constraint (6.9d).

Let $S_{il}(\theta)$ denote the set of tasks whose some task fraction is running at time θ, $\theta \in \{1, \dots, T_l\}$, on machine i. Consider machine $i^\star$. We construct a bipartite graph $G = (U \cup V, E)^2$ as follows. With a slight abuse of notations, for each time slot $\theta \in \{1, \dots, T_l\}$, we consider a node θ, and define $V = \{\theta | 1 \leq \theta \leq T_l - D_l\}$, and $U = \{\theta | T_l - D_l + 1 \leq \theta \leq T_l\}$. For any $s \in U$ and $t \in V$, we add an edge (s, t) if $h_{i^\star l}(s) + h_{i^\star l}(t) \geq m_{i^\star}$. This implies that if $h_{i^\star l}(s) + h_{i^\star l}(t) < m_{i^\star}$, then there is

[2] $G = (U \cup V, E)$ is a bipartite graph iff for any edge $e = (u, v) \in E$, we have $u \in U$ and $v \in V$.

no edge between s and t, and we can write

$$\big(\cup_{i\in\mathcal{M}} S_{il}(s)\big) \setminus \big(\cup_{i\in\mathcal{M}} S_{il}(t)\big) = \emptyset. \tag{6.11}$$

This is because otherwise SynchPack-1 would have scheduled the task(s) in $S_{i^\star l}(s)$ at time t (note that $t < s$).

Let $|\cdot|$ denote set cardinality (size). For any set of nodes $\tilde{U} \subseteq U$, we define set of its neighbor nodes as $N_{\tilde{U}} = \{t \in V | \exists\, s \in \tilde{U} : (s,t) \in E\}$. Note that, there are $T_l - D_l - |N_{\tilde{U}}|$ nodes in V which do not have any edge to some node in $\tilde{U}$. We consider two cases:

Case (i): There exists a set $\tilde{U}$ for which $|N_{\tilde{U}}| < |\tilde{U}|$. Consider a node $s \in \tilde{U}$ and a task with duration p running at time slot s. Let p_U denote the amount of time that this task is running on time slots of set U. Note that $p_U \geq 1$. By Equation (6.11), a task that is running at time s is also running at $T_l - D_l - |N_{\tilde{U}}|$ many other time slots whose corresponding nodes are in V.

$$p = T_l - D_l - |N_{\tilde{U}}| + p_U \leq D_l,$$

where the inequality is by Constraint (6.9c). Therefore

$$T_l \leq 2D_l + |N_{\tilde{U}}| - p_U < 2D_l + |\tilde{U}| \leq 3D_l.$$

Case (ii): For any $\tilde{U} \subseteq U$, $|\tilde{U}| \leq |N_{\tilde{U}}|$. Hence, $|V| \geq |U|$ which implies that $T_l \geq 2D_l$. Further, Hall's Theorem [175] states that a perfect matching[3] of nodes in U to nodes in V always exists in G in this case. The existence of such a matching then implies that any time slot $s \in (T_l - D_l, T_l]$ can be matched to a time slot $t_s \in (0, T_l - D_l]$ and $h_{i^\star l}(s) + h_{i^\star l}(t_s) \geq m_i$. This implies that

$$\sum_{s\in U}(h_{i^\star l}(s) + h_{i^\star l}(t_s)) \geq m_{i^\star} D_l \overset{(c)}{\geq} \sum_{t=1}^{T_l} h_{i^\star l}(t), \tag{6.12}$$

[3]A perfect matching in G (with size $|U|$) is a subset of E such that every node in set U is matched to one and only one node in set V by an edge in the subset.

where Inequality (c) is by Equation (6.10). From this, one can conclude that no non-zero task fraction $(k, j, i^\star, l')$, $i^\star$, $l' \leq l$ is processed at time slots $V' = V \setminus \cup_{s \in U}\{t_s\}$. This is because the right hand side of Inequality (c) is the total amount of task fractions that is processed up to time T_l. Hence, $V' = \emptyset$, since otherwise SynchPack-1 would have scheduled some of the tasks running at time slots of set U, at V'. We then can conclude that $T_l = 2D_l < 3D_l$. This completes the proof.

□ □

Recall that schedule $\bar{\mathcal{S}}$ is formed by stretching schedule $\mathcal{S}$ by factor $1/\lambda$. Let $\bar{C}_j^\lambda$ denote the completion time of job j in $\bar{\mathcal{S}}$. Next, we need to relate $\bar{C}_j^\lambda$ and $\tilde{C}_j$, the optimal solution to (LP1) for completion time of job j. For this purpose, we first make the following definition regarding schedule $\mathcal{S}$.

Definition 14. *We define $C_j(\alpha)$, for $0 < \alpha \leq 1$, to be the time at which α-fraction of job j is completed in schedule $\mathcal{S}$ (i.e., at least α-fraction of each of its tasks has been completed.).*

The following lemma shows the relationship between $C_j(\alpha)$ and $\tilde{C}_j$. The proof is provided in Section 6.9.2.

Lemma 23. $\int_{\alpha=0}^{1} C_j(\alpha) d\alpha \leq 3(1+\epsilon)\tilde{C}_j$

Now, we can show that the following lemma holds.

Lemma 24. $\mathbb{E}\left[\bar{C}_j^\lambda\right] \leq 6(1+\epsilon)\tilde{C}_j$.

Proof. Proof. The proof is based on Lemma 23 and taking expectation with respect to probability density function of λ. The details can be found in Section 6.9.3. □ □

In constructing $\bar{\mathcal{S}}$, we may shift scheduling time of some of the tasks on each machine to the left and construct a better schedule. Nevertheless, we have the performance guarantee of Theorem 11 even without this shifting.

Proof. Proof of Theorem 11. Let C_j denote the completion time of job j under SynchPack-1. Then

$$\mathbb{E}\Big[\sum_{j\in\mathcal{J}} w_j C_j\Big] \leq \mathbb{E}\Big[\sum_{j\in\mathcal{J}} w_j \bar{C}_j^{\lambda}\Big] \overset{(a)}{\leq} 6(1+\epsilon)\sum_{j\in\mathcal{J}} w_j \tilde{C}_j \overset{(b)}{\leq} 6(1+\epsilon)\sum_{j\in\mathcal{J}} w_j C_j^{\star},$$

where (a) is by Lemma 24, and (b) is by Lemma 21. In Section 6.10, we discuss how to derandomize the random choice of $\lambda \in (0,1]$, which is used to construct schedule $\bar{\mathcal{S}}$ from schedule $\mathcal{S}$. So the proof is complete. □ □

6.4 Scheduling When Migration is not Allowed

The algorithm in Section 6.3 is preemptive, and tasks can be migrated across the machines in the same placement set. Implementing such an algorithm can be complex and costly in practice. In this section, we consider the case that migration of tasks among machines is not allowed. We propose a non-preemptive scheduling algorithm for this case. Using the 3-field notation, this case is represented by PRP$|\ |\sum_j w_j C_j$. We also show that its solution provides a bounded solution for the case that preemption of tasks (in the same machine, without migration) is allowed (PRP$|$pmtn$|\sum_j w_j C_j$).

Our algorithm is based on a relaxed LP which is very similar to (LP1) of Section 6.3, however a different constraint is used to ensure that each task is scheduled entirely by the end point of some time-interval of a machine. Next, we introduce this LP and describe how to generate a non-preemptive schedule based on its solution.

6.4.1 Relaxed Linear Program (LP2)

We partition the time horizon into intervals $(d_{l-1}, d_l]$ for $l = 0, ..., L$, as defined in (6.5) by replacing ϵ by 1. Define 0-1 variable z_{kj}^{il} to indicate whether task (k, j) is completed on machine i by the end-point of interval l, i.e., by d_l. Note that the interpretation of variables z_{kj}^{il} is slightly different from their counterparts in (LP1). By relaxing integrality of z variables, we formulate

(LP2):

$$\min \sum_{j \in \mathcal{J}} w_j C_j \qquad (\mathbf{LP2}) \tag{6.13a}$$

$$z_{kj}^{il} = 0 \ \text{ if } p_{kj}^{i} > d_l, \ \ j \in \mathcal{J}, \ k \in \mathcal{K}_j, \ i \in \mathcal{M}_{kj}, \ l = 0, \ldots, L \tag{6.13b}$$

$$\text{Constraints (6.9b)–(6.9h)} \tag{6.13c}$$

Note that Constraint (6.13b) allows z_{kj}^{il} to be positive only if the end point of the l-th interval is at least as long as task (k, j)'s processing time on machine $i \in \mathcal{M}_{kj}$. We would like to emphasize that this is a valid constraint for both the preemptive and non-preemptive cases when migration is not allowed. We will see shortly how this constraint helps us construct our non-preemptive algorithm. We interpret *fractional* values of z_{kj}^{il} as the fraction of task (k, j) that is processed in interval l of machine i (as in Section 6.3).

6.4.2 Scheduling Algorithm: SynchPack-2

Our non-preemptive algorithm, which we refer to as SynchPack-2, has three main steps:

Step 1: Solve (LP2). We first solve the linear program (LP2) to obtain the optimal solution of $\{z_{kj}^{il}\}$ denoted by $\{\tilde{z}_{kj}^{il}\}$.

Step 2: Apply Slow-Motion. Before constructing the actual schedule of tasks, the algorithm applies the Slow-Motion technique (see Section 6.3.2). We pause here to clarify the connection between $\tilde{z}_{kj}^{il}$ and those obtained after applying Slow-Motion which we denote by $\bar{z}_{kj}^{il}$, below.

Recall that $\tilde{z}_{kj}^{il}$ is the fraction of task (k, j) that is scheduled in interval l of machine i in the optimal solution to (LP2), and Δ_l is the length of the l-th interval. Also, recall that $\tilde{Z}_{kj}^{i} = \sum_{l=0}^{L} \tilde{z}_{kj}^{il}$ is the total task fraction to be scheduled on machine i corresponding to task (k, j). Similarly, we define $\bar{\Delta}_l$ and $\bar{d}_l$ to be the length and the end point of the l-th interval after applying the Slow-Motion using a stretch parameter $\lambda \in (0, 1]$, respectively. Therefore,

$$\bar{\Delta}_l = \frac{\Delta_l}{\lambda}, \ \ \bar{d}_l = \frac{d_l}{\lambda}. \tag{6.14}$$

Further, we define $\bar{z}_{kj}^{il}$ to be the fraction of task (k, j) to be scheduled during the l-th interval on machine i after applying Slow-Motion. Then it holds that,

$$\bar{z}_{kj}^{il} = \begin{cases} \frac{\tilde{z}_{kj}^{il}}{\lambda}, & \text{if } \sum_{l\prime=0}^{l} \frac{\tilde{z}_{kj}^{il'}}{\lambda} < \tilde{Z}_{kj}^{i} \\ \max\left\{0, \left(\tilde{Z}_{kj}^{i} - \sum_{l\prime=0}^{l-1} \frac{\tilde{z}_{kj}^{il'}}{\lambda}\right)\right\}, & \text{otherwise.} \end{cases} \tag{6.15}$$

To see (6.15), note that in Slow-Motion, both variables and intervals are stretched by factor $1/\lambda$, and after stretching, the machine is left idle if it has already processed its total task fraction completely. Hence, as long as $\tilde{Z}_{kj}^{i}$ fraction of task (k, j) is not completely processed by the end of the l-th interval in the stretched solution, it is processed for $\tilde{z}_{kj}^{il} p_{kj}^{i}/\lambda$ amount of time in the l-th interval of length $\bar{\Delta}_l = \Delta_l/\lambda$. Hence $\bar{z}_{kj}^{il} = \tilde{z}_{kj}^{il}/\lambda$. Now suppose $l^\star$ is the first interval for which $\sum_{l\prime=0}^{l^\star} \tilde{z}_{kj}^{il'}/\lambda \geq \tilde{Z}_{kj}^{i}$. Then, the remaining processing time of task (k, j) to be scheduled in the $l^\star$-th interval of machine i in the stretched schedule is $p_{kj}^{i}(\tilde{Z}_{kj}^{i} - \sum_{l\prime=0}^{l^\star-1} \bar{z}_{kj}^{il'}) = p_{kj}^{i}(\tilde{Z}_{kj}^{i} - \sum_{l\prime=0}^{l^\star-1} \tilde{z}_{kj}^{il'}/\lambda) > 0$. Therefore, the second part of (6.15) holds for $l^\star$, and for intervals $l > l^\star$, $\bar{z}_{kj}^{il}$ will be zero, since $\tilde{Z}_{kj}^{i} - \sum_{l\prime=0}^{l-1} \tilde{z}_{kj}^{il'}/\lambda \leq 0$. Observe that $\sum_{i \in \mathcal{M}_{kj}} \sum_{l=0}^{L} \bar{z}_{kj}^{il} = 1$.

Step 3: Construct a non-preemptive schedule. Note that according to variables $\bar{z}_{kj}^{il}$, a task possibly is set to get processed in different intervals and machines. The last step of SynchPack-2 is the procedure of constructing a non-preemptive schedule using these variables. This procedure involves 2 substeps: (1) *mapping of tasks to machine-intervals*, and (2) *non-preemptive scheduling of tasks mapped to each machine-interval using a greedy scheme*. We now describe each of these substeps in detail.

Substep 3.1: Mapping of tasks to machine-intervals. For each task (k, j), the algorithm uses a mapping procedure to find *a machine and an interval in which it can schedule the task entirely in a non-preemptive fashion.* The mapping procedure is based on constructing a weighted bipartite graph $\mathcal{G} = (U \cup V, E)$, followed by an *integral matching* of nodes in U to nodes in V on edges with non-zero weights, as described below:

(i) *Construction of Graph $\mathcal{G} = (U \cup V, E)$:* For each task (k, j), $j \in \mathcal{J}$, $k \in \mathcal{K}_j$, we consider a node in U. Therefore, there are $\sum_{j \in \mathcal{J}} |\mathcal{K}_j|$ nodes in U. Further, $V = \cup_{i \in \mathcal{M}} V_i$, where V_i is the set of nodes that we add for machine i to represent intervals. To construct graph $\mathcal{G}$, we start from the first machine, say machine i, and sort tasks in non-increasing order of their volume $v^i_{kj} = a_{kj} p^i_{kj}$ in machine i. Let N_i denote the number of tasks on machine i with nonzero volumes. Without loss of generality, suppose

$$v^i_{k_1 j_1} \geq v^i_{k_2 j_2} \geq \dots v^i_{k_{N_i} j_{N_i}} > 0. \tag{6.16}$$

For each interval l, we consider $\lceil \bar{z}^{il} \rceil = \lceil \sum_{j \in J} \sum_{k \in \mathcal{K}_j} \bar{z}^{il}_{kj} \rceil$ (recall the definition of $\bar{z}^{il}_{kj}$ in (6.15)) consecutive nodes in V_i which we call *copies of interval l*.

Starting from the first task in the ordering (6.16), we draw edges from its corresponding node in U to the interval copies in V_i in the following manner. Assume we reach at task (k, j) in the process of adding edges. For each interval l, if $\bar{z}^{il}_{kj} > 0$, first set $R = \bar{z}^{il}_{kj}$. Consider the first copy of interval l for which the total weight of its current edges is strictly less than 1 and set W to be its total weight. We draw an edge from the node of task (k, j) in U to this copy node in V_i, and assign a weight equal to $\min\{R, 1 - W\}$ to this edge. Then we update $R \leftarrow R - \min\{R, 1 - W\}$, consider the next copy of interval l, and apply the same procedure, until $R = 0$ (or equivalently, the sum of edge weights from node (k, j) to copies of interval l becomes equal to $\bar{z}^{il}_{kj}$). We use w^{ilc}_{kj} to denote the weight of edge that connects task (k, j) to copy c of interval l of machine i, and if there is no such edge, $w^{ilc}_{kj} = 0$. We then move to the next machine and apply the similar procedure, and so on. See Figure 6.3 for an illustrative example.

Note that in $\mathcal{G}$, the weight of any node $u \in U$ (the sum of weights of its edges) is equal to 1 (since $\sum_{l=0}^{L} \bar{z}^{il}_{kj} = 1$, for any task (k, j)), while the weight of any node $v \in V$ is at most 1.

(ii) *Integral Matching:* Finally, we find an integral matching on the non-zero edges of $\mathcal{G}$, such that each non-zero task is matched to some interval copy. As we will show shortly in Section 6.4.3, we can always find an integral matching of size $\sum_{j \in \mathcal{J}} |\mathcal{K}_j|$, the total number of tasks, in $\mathcal{G}$, in polynomial time, in which each task is matched to a copy of some interval.

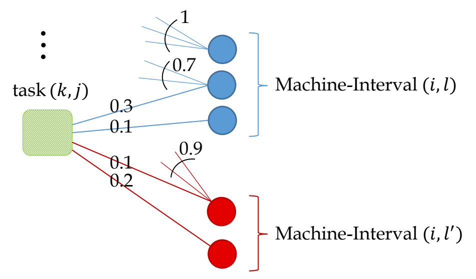


Figure 6.3: An illustrative example for construction of graph $\mathcal{G}$ in Substeb 3.1. Task (k, j) requires $\bar{z}_{kj}^{il} = 0.4$ and $\bar{z}_{kj}^{il'} = 0.3$. When we reach at task (k, j), the total weight of the first copy of interval l is 1 and that of its second copy is 0.7. Also, the total weight of the first copy of interval l' is 0.9. Hence, the procedure adds 2 edges to copies of interval l with weights 0.3 and 0.1, and 2 edges to copies of interval l' with weights 0.1 and 0.2.

A pseudocode for the mapping procedure can be found in Section 6.15.

Substep 3.2: Greedy packing of tasks in machine-intervals. We utilize a greedy packing to schedule all the tasks that are mapped to a machine-interval *non-preemptively*. More precisely, on each machine, the greedy algorithm starts from the first interval and considers an arbitrary ordered list of its corresponding tasks. Starting from the first task, the algorithm schedules it, and moves to the second task. If the machine has sufficient capacity, it schedules the task, otherwise it checks the next task and so on. Once it is done with all the tasks of the first interval, it considers the second interval, applies the similar procedure, and so on. We may also shift back future tasks' schedules as far as the machine capacity allows.

Note that this greedy algorithm is simpler than the one described in Section 6.3, since it does not need to consider requirement (ii) of Section 6.2 as here each task only appears in one feasible machine.

As we prove in the next section, we can bound the total volume of tasks mapped to interval l on machine i in the mapping phase by $m_i\bar{\Delta}_l$. Furthermore, by Constraint (6.13b) and the fact that the integral matching in Substep 3.1 was constructed on non-zero edges, the processing time of any task mapped to an interval is not greater than the interval's end point, which is twice the interval length. Hence, we can bound the completion time of each job and find the approximation ratio that our algorithm provides. A pseudocode for the SynchPack-2 algorithm can be found in Section 6.15.

6.4.3 Performance Guarantee

In this section, we analyze the performance of our non-preemptive algorithm SynchPack-2. The main result of this section is as follows:

Theorem 12. *The scheduling algorithm SynchPack-2 is a* 24*-approximation algorithm for the problem of parallel-task jobs scheduling with packing and placement constraints, when preemption and migration is not allowed.*

Since the constraints of (LP2) also hold for the preemptive case when migration is not allowed, the optimal solution of this case is also lower bounded by the optimal solution to the LP. Therefore, the algorithm' solution is also a bounded solution for the case that preemption is allowed (while still migration is not allowed).

Corollary 4. *The scheduling algorithm SynchPack-2, in Section 6.4.2, is a* 24*-approximation algorithm for the problem of parallel-task jobs scheduling with packing and placement constraints, when preemption is allowed and migration is not.*

The rest of this section is devoted to the proof of Theorem 12. With a minor abuse of notation, we use $\tilde{C}_{kj}$ and $\tilde{C}_j$ to denote the completion time of task (k, j) and job j, respectively, in the optimal solution to (LP2). Also, let $C^\star_{kj}$ and $C^\star_j$ denote the completion time of task (k, j) and job j, respectively, in the optimal non-preemptive schedule. We can bound the optimal value of (LP2) as stated below. The proof is provided in Section 6.11.1.

Lemma 25. $\sum_{j=1}^{N} w_j \tilde{C}_j \leq \sum_{j=1}^{N} w_j C^\star_j = OPT.$

Definition 15. *Given* $0 < \alpha \leq 1$*, define* $\hat{C}_j(\alpha)$ *to be the starting point of the earliest interval* l *for which* $\alpha \leq \tilde{x}_{jl}$ *in solution to (LP2).*

Note that $\hat{C}_j(\alpha)$ is slightly different from Definition 14, as we do not construct an actual schedule yet. We then have the following corollary which is a counterpart of Lemma 23. See Section 6.11.2 for the proof.

Corollary 5. $\int_{\alpha=0}^{1} \hat{C}_j(\alpha) d\alpha = \tilde{C}_j$

Consider the mapping procedure where we construct bipartite graph $\mathcal{G}$ and match each task to a copy of some machine-interval. Below, we state a lemma which ensures that indeed we can find an integral (i.e. 0 or 1) matching in $\mathcal{G}$. The proof can be found in Section 6.11.3.

Lemma 26. *Consider graph $\mathcal{G}$ constructed in the mapping procedure. There exists an integral matching on the nonzero edges of $\mathcal{G}$ in which each task is matched to some interval copy. Further, this matching can be found in polynomial time.*

Let $\mathcal{V}_{il}$ denote the total volume of the tasks mapped to all the copies of interval l of machine i. The following lemma bounds $\mathcal{V}_{il}$ whose proof is provided in Section 6.11.4.

Lemma 27. *For any machine-interval (i, l), we have*

$$\mathcal{V}_{il} \leq \bar{d}_l m_i + \sum_{j \in \mathcal{J}} \sum_{k \in \mathcal{K}_j} v^i_{kj} \bar{z}^{il}_{kj}. \tag{6.17}$$

Note that the second term in the right side of (6.17) can be bounded by $\bar{d}_l m_i$ which results in the inequality $\mathcal{V}_{il} \leq 2\bar{d}_l m_i$. However, the provided bound is tighter and allows us to prove a better bound for the algorithm. We next show that, using the greedy packing algorithm, we can schedule all the tasks of an interval l in a bounded time.

In the case of packing single tasks in a single machine, the greedy algorithm by [167] is known to provide a 2-approximation solution for minimizing makespan. The situation is slightly different in our setting as we require to bound the completion time of the last task as a function of the total volume of tasks, when the maximum duration of all tasks in each interval is bounded. We state the following lemma and its proof in Section 6.11.5 for completeness.

Lemma 28. *Consider a machine with capacity* 1 *and a set of tasks $J = \{1, 2, \ldots, n\}$. Suppose each task j has size $a_j \leq 1$, processing time $p_j \leq 1$, and $\sum_{j \in J} a_j p_j \leq v$. Then, we can schedule all the tasks within the interval* $(0, 2\max\{1, v\}]$ *using the greedy algorithm.*

Now consider a machine-interval (i, l). Note that Lemma 27 bounds the total volume of tasks while Constraint (6.13b) ensures that duration of each task is less than d_l. Thus, by applying

Lemma 28 on the normalized instance, in which size and length of tasks are normalized by m_i and d_l, respectively, we guarantee that we can schedule all the task within a time interval of length $2d_l + 2\sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}_j} v^i_{kj}\bar{z}^{il}_{kj}/m_i$. Moreover, the factor 2 is tight as stated in the following lemma whose proof can be found in Section 6.11.6.

Lemma 29. *We need an interval of length at least* $2\max(1,v)$ *to be able to schedule any list of tasks as in Lemma 28 using any algorithm.*

Hence, Lemmas 28 and 29 imply that applying the greedy algorithm to schedule the tasks of each machine-interval, provides a tight bound with respect to the total volume of tasks in that machine-interval. Let C_{kj} denote the completion time of task (k, j) under SynchPack-2. Then we have the following lemma whose proof can be found in Section 6.11.7.

Lemma 30. *Suppose that task* (k, j) *is mapped to the* l*-th interval of machine i at the end of Substep 3.1. Then,* $C_{kj} \le 6\bar{d}_l$.

Proof. Proof of Theorem 12. Let l denote the end point of the interval in which task (k, j) has the last non-zero fraction according to $\bar{z}^{il}_{kj}$. Then,

$$\bar{d}_l = 2^l/\lambda \overset{(\star)}{\le} 2\hat{C}_j(\lambda)/\lambda. \tag{6.18}$$

First note that ϵ is replaced by 1 in Equation (6.5). Further, Inequality ($\star$) follows from the definition of $\hat{C}_j(\lambda)$ (Definition 15), and the fact that d_l's are multiplied by $1/\lambda$. Therefore, $\hat{C}_j(\lambda)/\lambda$ is the start point of the interval in which job j is completed, and, accordingly, $2\hat{C}_j(\lambda)/\lambda$ is the end point of that interval. Thus, $2^l/\lambda$, the end point of the interval in which task (k, j) is completed, has to be at most $2\hat{C}_j(\lambda)/\lambda$, the end point of the interval in which job j is completed.

Let C_{kj} and C_j be the completion time of task (k, j) and job j under SynchPack-2. Recall that in the mapping procedure, we only map a task to some interval l' in which part of the task is assigned to that interval after Slow-Motion applied (in other words, $\bar{z}^{il'}_{kj} > 0$). Thus, task (k, j) that has its last non-zero fraction in interval l (by our assumption) is mapped to some interval $l' \le l$, because $\bar{z}_{kj}^{il''} = 0$ for intervals $l'' > l$. Suppose task (k_j, j) is the last task of job j and finishes in interval l_j in our non-preemptive schedule. Then, by Lemma 30 and Equation (6.14), we have $C_j = C_{i_j j} \leq 6\bar{d}_l = \frac{6}{\lambda}2^{l_j}$. Recall that $\tilde{C}_j$ denotes the completion time of job j in an optimal solution of (LP2). Hence,

$$\mathbb{E}\Big[\sum_{j\in\mathcal{J}} w_j C_j\Big] \leq \mathbb{E}\Big[\sum_{j\in\mathcal{J}} w_j \frac{6}{\lambda}2^{l_j}\Big] \overset{(a)}{\leq} 12 \times \mathbb{E}\Big[\sum_{j\in\mathcal{J}} w_j \hat{C}_j(\lambda)/\lambda\Big]$$
$$\overset{(b)}{=} 12 \times \sum_{j\in\mathcal{J}} w_j \int_{\lambda=0}^{1} \frac{\hat{C}_j(\lambda)}{\lambda} 2\lambda d\lambda \overset{(c)}{\leq} 24 \times \sum_{j\in\mathcal{J}} w_j \tilde{C}_j,$$

where in the above, (a) is by the second part of (6.18) for $l = l_j$, (b) is by definition of expectation with respect to λ, with pdf $f(\lambda) = 2\lambda$, and (c) is by Corollary 5. Using the above inequality and Lemma 25,

$$\mathbb{E}\Big[\sum_{j\in\mathcal{J}} w_j C_j\Big] \leq 24 \times \sum_{j\in\mathcal{J}} w_j C_j^\star = 24 \times \text{OPT}. \tag{6.19}$$

By applying de-randomization procedure (see Section 6.10), we can find $\lambda = \lambda^\star$ in polynomial time for which the total weighted completion time is less that its expected value in (6.19). This completes the proof of Theorem 12. □ □

6.5 Special Case: Preemption and Single-Machine Placement set

In previous sections, we studied the parallel-task job scheduling problem for both cases when migration of tasks (among machines in its placement set) is allowed or not, and provided $(6 + \epsilon)$ and 24 approximation algorithms, respectively. In this section, we consider a special case when only one machine is in the placement set of each task (e.g., it is the only machine that has the required data for processing the task), and preemption is allowed. Using the 3-field notation, this case is represented by PDP|pmtn|$\sum_j w_j C_j$.

Corollary 6. *Consider the parallel-task job scheduling problem when there is a specific machine to process each task and preemption is allowed. For any $\epsilon > 0$, the sum of the weighted completion times of jobs under SynchPack-1, in Section 6.3.2, is at most $(4 + \epsilon) \times OPT$.*

Proof. Proof. The proof is straight forward and similar to proof of Theorem 11. Specifically, the factor 3 needed to bound the solution of the greedy policy is reduced to 2 due to the fact that placement constraint is not needed to be enforced here, since there is only one machine for each task. □ □

We can show that there is a slightly better approximation algorithm to solve the problem in this special case, that has an approximation ratio 4. The algorithm uses a relaxed LP, based on linear ordering variables (e.g., see [84, 75, 77]) to find an efficient ordering of jobs. Then it applies a simple list scheduling to pack their tasks in machines subject to capacity constraints. The details are as follows.

6.5.1 Relaxed Linear Program (LP3)

Note that each task has to be processed in a specific machine. Each job consists of up to M (number of machines) different tasks. We use $\mathcal{M}_j$ to denote the set of machines that have tasks for job j. Task i of job j, denoted as task (i, j), requires a specific amount a_{ij} of machine i's resource ($a_{ij} \leq m_i$) for a specific time duration $p_{ij} > 0$. We also define its volume as $v_{ij} = a_{ij}p_{ij}$. The results also hold in the case that a job has multiple tasks on the same machine.

For each pair of jobs, we define $\delta_{jj'} \in \{0, 1\}$ such that $\delta_{jj'} = 1$ if job j is completed before job j', and $\delta_{jj'} = 0$ otherwise. Note that by the synchronization constraint (6.1), the completion of a job is determined by its last task. If both jobs finish at the same time, we set either one of $\delta_{jj'}$ or $\delta_{j'j}$ to 1 and the other one to 0, arbitrarily. By relaxing the integral constraint on binary variables, we formulate the following LP:

$$\min \sum_{j \in \mathcal{J}} w_j C_j \qquad \textbf{(LP3)} \tag{6.20a}$$

$$m_i C_j \geq v_{ij} + \sum_{j' \in \mathcal{J}, j' \neq j} v_{ij'} \delta_{j'j}, \;\; j \in \mathcal{J}, i \in \mathcal{M}_j \tag{6.20b}$$

$$C_j \geq p_{ij}, \;\; j \in \mathcal{J}, i \in \mathcal{M}_j \tag{6.20c}$$

$$\delta_{jj'} + \delta_{j'j} = 1, \;\; j \neq j', \; j, j' \in \mathcal{J} \tag{6.20d}$$

$$\delta_{jj'} \geq 0, \;\; j, j' \in \mathcal{J} \tag{6.20e}$$

Recall the definition of job completion time C_j and task completion time C_{ij} in Section 6.2. In (LP3), (6.20b) follows from the definition of $\delta_{jj'}$, and the fact that the tasks which need to be served on machine i are processed by a single machine of capacity m_i. It states that the total volume of tasks that can be processed during the time period $(0, C_j]$ by machine i is *at most* $m_i C_j$. This total volume is given by the right-hand-side of (6.20b) which basically sums the volumes of the tasks on machine i that finish before job j finishes its corresponding tasks at time C_j, plus the volume of task (i, j) itself. Constraint (6.20c) is due to the fact that $C_j \geq C_{ij}$ and each task cannot be completed before its processing time p_{ij}. (6.20d) indicates that for each two jobs, one precedes the other. Further, we relax the binary ordering variables to be fractional in (6.20e).

Note that the optimal solution to (LP3) might be an infeasible schedule as (LP3) replaces the tasks by sizes of their volumes and it might be impossible to pack the tasks in a way that matches the obtained completion times from (LP3).

Remark 4. *(LP3) can be easily modified to allow each job to have multiple tasks on the same machine. We omit the details to focus on the main ideas.*

6.5.2 Scheduling Algorithm: SynchPack-3

The SynchPack-3 algorithm has two steps:

Step 1: Solve (LP3) to find an ordering of jobs. Let $\tilde{C}_j$ denote the optimal solution to (LP3) for completion time of job $j \in \mathcal{J}$. We order jobs based on their $\tilde{C}_j$ values in a nondecreasing order. Without loss of generality, we re-index the jobs such that

$$\tilde{C}_1 \leq \tilde{C}_2 \leq ... \leq \tilde{C}_N. \tag{6.21}$$

Ties are broken arbitrarily.

Step 2: List scheduling based on the obtained ordering. For each machine i, the algorithm maintains a list of tasks such that for every two tasks (i, j) and (i, j') with $j < j'$ (according to ordering (6.21)), task (i, j) appears before task (i, j') in the list. On machine i, the algorithm scans the list starting from the first task. It schedules a task (i, j) from the list if the machine has sufficient remaining resource to accommodate it. Upon completion of a task, the algorithm preempts the schedule, removes the completed task from the list and updates the remaining processing time of the tasks in the list, and starts scheduling the tasks in the updated list. Observe that this list scheduling is slightly different from the greedy scheme used in SynchPack-1. A pseudocode for the algorithm can be found in Section 6.16.

6.5.3 Performance Guarantee

Theorem 13. *The scheduling algorithm SynchPack-3 is a* 4*-approximation algorithm for the problem of parallel-task jobs scheduling with packing and single-machine placement constraints.*

The proof of the theorem, and any supporting lemmas, is presented in Section 6.12.

6.6 Complexity of Algorithms

The complexity of our algorithms is mainly dominated by solving their corresponding LPs, which can be solved in polynomial time using efficient linear programming solvers. The rest of the operations have low complexity and can be parallelized on the machines. We have provided a detailed discussion of the complexity in Section 6.8.

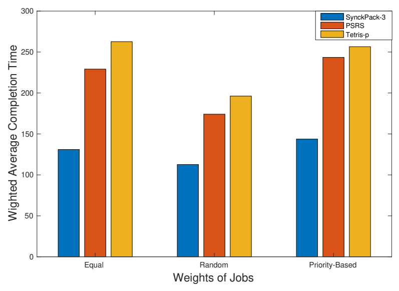
6.7 Evaluation Results

In this section, we evaluate the performance of our algorithms using a real traffic trace from a large Google cluster [151], and compare to prior algorithms. The original data set only contains the machine to which each task is assigned by the resource manager, and the information regarding the placement constraints (data locality) is missing. The setting is then similar to our model for preemptive algorithm SynchPack-3 in Section 6.5. To incorporate placement constraints, we modify the data set as follows. For each task, we randomly choose 3 machines and assume that processing time of the task on these machines is equal to the processing time given in the data set. We allow the task to be scheduled on other machines; however, its processing time will be penalized by a factor $\alpha > 1$. This is consistent with the data locality models in previous work (e.g. [145, 163]). The details of the data set can be found in Section 6.13.

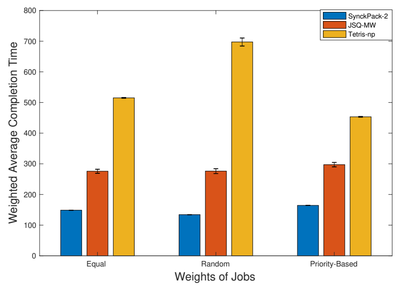
We consider three prior algorithms, PSRS [169], Tetris [145], and JSQ-MW [163] to compare with our algorithms SynchPack-2 and SynchPack-3. PSRS is a preemptive algorithm for the parallel task scheduling problem (see Section 6.1.1) on a single machine. Tetris is a heuristic that schedules tasks on each machine according to an ordering based on their scores (Section 6.1.1). In our evaluations, we consider two versions of Tetris, preemptive (Tetris-p) and non-preemptive (Tetris-np). Finally, Join-the-Shortest-Queue routing with Max Weight scheduling (JSQ-MW) is a non-preemptive algorithm in presence of data locality (Section 6.1.1). An overview of these algorithms can be found in Section 6.13.

6.7.1 Results in Offline Setting

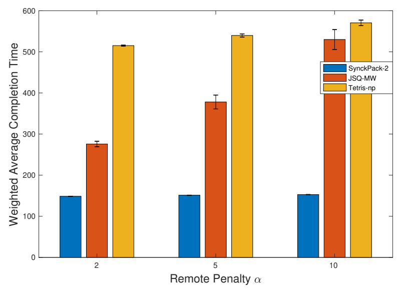
We use SynchPack-3, Tetris-p, and PSRS to schedule tasks of the original data set preemptively, and use SynchPack-2, Tetris-np, and JSQ-MW to schedule tasks of the modified data set (with placement constraints) non-preemptively. We then compare the weighted average completion time of jobs, $\sum_j w_j C_j / \sum_j w_j$, under these algorithms for the three weight cases, i.e. equal, random, and priority-based weights. Note that weighted average completion time is equivalent to the total



(a) Performance of SynchPack-3, Tetris-p, and PSRS for different weights.

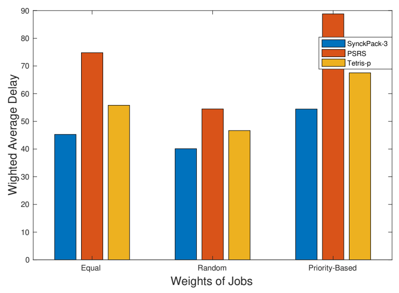


(b) Performance of SynchPack-2, Tetris-np, and JSQ-MW for different weights and remote penalty $\alpha = 2$.

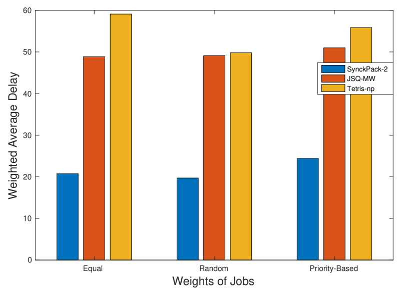


(c) Performance of SynchPack-2, Tetris-np, and JSQ-MW for different remote penalties and equal weights.

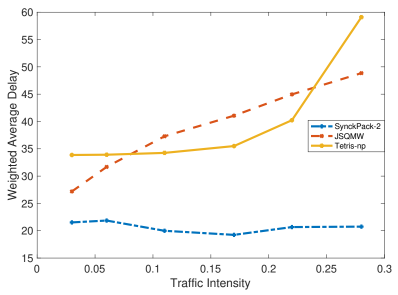
Figure 6.4: Performance of algorithms in the offline setting.



(a) Performance of SynchPack-3, Tetris-p, and PSRS for different weights.



(b) Performance of SynchPack-2, Tetris-np, and JSQ-MW for different weights.



(c) Performance of SynchPack-2, Tetris-np, and JSQ-MW for different traffic intensities.

Figure 6.5: Performance of algorithms in the online setting.

weighted completion time (up to the normalization $\sum_j w_j$). We first report the ratio between the total weighted completion time obtained from SynchPack-2 (for $\alpha = 2$) and SynchPack-3 and their corresponding optimal value of their relaxed LPs (6.13) and (6.20) (which are lower bounds on the optimal total weighted competition times) to verify Theorem 12 and 13. Table 6.1 shows this performance ratio for the 3 cases of job weights. All ratios are within our theoretical results of 24 and 4. In fact, the approximation ratios are much smaller.

Table 6.1: Performance ratio of SynchPack-3 with respect to (LP3), and SynchPack-2 with respect to (LP2)

Jobs' Weights	Equal	Random	Priority-Based
Ratio for SynchPack-2	2.87	2.90	2.98
Ratio for SynchPack-3	1.34	1.35	1.31

Figure 6.4a shows the performance of SynchPack-3, Tetris-p, and PSRS in the offline setting. As we see, SynchPack-3 outperforms the other two algorithms in all the cases and performance gain varies from 33% to 132%. Further, Figure 6.4b depicts performance of SynchPack-2, Tetris-np, and JSQ-MW for different weights, when $\alpha = 2$. The performance gain of SynchPack-2 varies from 81% to 420%. Figure 6.4c shows the effect of remote penalty α in the performance of SynchPack-2, Tetris-np, and JSQ-MW. As we see, SynchPack-2 outperforms the other algorithms by 85% to 273%

6.7.2 Results in Online Setting

In the online setting, jobs arrive dynamically over time, according to the arrival time information in the data set, and we are interested in the weighted average *delay* of jobs. The delay of a job is measured from the time that it arrives to the system until its completion. See Section 6.13 for details on implementation of the algorithms in the online setting.

Figure 6.5a shows the performance results, in terms of the weighted average *delay* of jobs, under SynchPack-3, Tetris-p, and PSRS. Performances of Tetris-p is worse than our algorithm by 11% to 27%, while PSRS presents the poorest performance and has 36% to 65% larger weighted average delay compared to SynchPack-3. Moreover, performance of SynchPack-2, Tetris-np, and JSQ-MW for different weights is depicted in Figure 6.5b. As we see, SynchPack-2 outperforms the other two algorithms in all the cases and performance gain varies from 109% to 189%. Further, by multiplying arrival times by constant values we can change the traffic intensity and study its effect on algorithms' performance. Figure 6.5c shows the results for equal job weights. As we can see, SynchPack-2 outperforms the other algorithms and the performance gain increases as traffic intensity grows.

6.8 Complexity of Algorithms

The linear program (LP1) in (6.9) has at most $KNML + NL + N$ variables (K is the maximum number of tasks a job has.), which is polynomially bounded in the problem's input size. The number of constraints is also polynomially bounded. Hence, it can be solved in polynomial time using efficient linear programming solvers. The complexity of SynchPack-1 is mainly determined by solving (LP1). The complexity of Slow-Motion step is very low and can be parallelized in different machines, namely, $O(KNL)$ on each machine, and $O(KNLM)$ in total. The complexity of the greedy list scheduling – upon arrival or departure of a task fraction– is at most the length of the list (equal to the number of incomplete task fractions which is initially equal to $O(KNLM)$) times the number of machines M.

Mapping procedure is the extra step for SynchPack-2. The complexity of this step is also polynomially bounded in input size and is $O(K^2N^2ML)$. $O(KN + ML)$ is used for constructing the graph as there are $O(KN)$ nodes on one side (number of all the tasks), $O(KN + ML)$ on the other side (number of all machine-interval copies), and it takes $O(KNML)$ to create edges (each task has at most 2 edges to copies of each machine-interval.). Further, finding an integral matching from the fractional matching takes $O(K^2N^2ML)$. The greedy algorithm in SynchPack-2 can be parallelized on the machines and takes $O(KN)$ in total.

Similarly, the complexity of SynchPack-3 is mainly dominated by solving (LP3) to find an appropriate ordering of jobs. The relaxed linear program (LP3) has $O(N^2)$ variables and $O(N^2 + MN)$ constraints and can be solved in polynomial time using efficient linear programming solvers. Note that the job ordering is the same on all the machines and they simply list-schedule their tasks respecting this ordering, independently of other machines. The complexity of the list scheduling is less than the one used in SynchPack-2 and is at most the length of the list, which is equal to the number of incomplete tasks.

Further, we would like to emphasize that in all the algorithms the corresponding linear program (LP) is solved *only once* at the beginning of the algorithm.

For the simulations, we used Gurobi software [176] to solve (LP2) and (LP3) in the simulations. On a desktop PC, with 8 Intel CPU core $i7 - 4790$ processors @ 3.60 GHz and 32.00 GB RAM, the average time it took to solve (LP1) was 145 seconds under offline setting. For purpose of comparison, the maximum job completion and the weighted average completion time time under our algorithm are 4.3×10^4 seconds and 8.6×10^3 seconds, respectively, for the case of priority-based weights. For solving (LP3), the average time it took was 435 seconds under offline setting, while the maximum job completion time and the weighted average completion time under our algorithm are 4.8×10^4 seconds and 10^4 seconds, respectively for the case of priority-based weights for $\alpha = 2$. We note that solving the LPs can be done much faster using the powerful computing resources in today's data centers.

6.9 Proofs Related to SynchPack-1

6.9.1 Proof of Lemma 21

Consider an optimal solution to the task scheduling problem with packing and synchronization constraints. Define $\hat{C}^\star_{kj}$ (similarly, $\hat{C}^\star_j$) to be the left point of the interval in which task (k, j) (similarly, job j) completes in the optimal schedule. Clearly, $\hat{C}^\star_j \le C^\star_j$. We set $z^{il\star}_{kj}$ equal to the fraction of task (k, j) that is scheduled in interval l on machine i. Also, we set $x^\star_{j,l}$ to be one for the last interval that some task of job j is running in the optimal schedule and to be zero for other intervals. Obviously, $\hat{C}^\star_j = \sum_{l=0}^{L} d_{l-1} x^\star_{jl}$. It is easy to see that the set of values $\hat{C}^\star_j$, $z^{il\,\star}_{kj}$, and $x^\star_{j,l}$ satisfies all the constraints of (LP3). Therefore, $\sum_{j=1}^{N} w_j \tilde{C}_j \le \sum_{j=1}^{N} w_j \hat{C}^\star_j \le \sum_{j=1}^{N} w_j C^\star_j$.

6.9.2 Proof of Lemma 23

Recall that τ_l is the time that all the task fractions (k, j, i, l'), for $l' \le l$, complete in schedule $\mathcal{S}$. Let α_l be the fraction of job j that is completed by τ_l.

Note that as we schedule all the task fractions (k, j, i, l'), for $l' \le l$ and possibly some other task fractions, we have,

$$\alpha_l \ge \sum_{l'=0}^{l} \tilde{x}_{jl'}. \tag{6.22}$$

We define $y_{jl} = \alpha_l - \alpha_{l-1}$. Note that $\sum_{l=0}^{L} y_{jl} = 1$. Moreover, $C_j(\alpha) \le 3d_l$ for $\alpha \in (\alpha_{l-1}, \alpha_l]$. The factor 3 comes from Lemma 22. Therefore:

$$
\begin{aligned}
\int_0^1 C_j(\alpha)d\alpha &= \sum_{l=0}^{L}\int_{\alpha_{l-1}}^{\alpha_l} C_j(\alpha)d\alpha \le \sum_{l=0}^{L}(\alpha_l - \alpha_{l-1}) \times 3d_l \\
&\overset{(a)}{=} 3(1+\epsilon)\sum_{l=0}^{L} y_{jl}d_{l-1} \overset{(b)}{\le} 3(1+\epsilon)\sum_{l=0}^{L}\tilde{x}_{jl}d_{l-1} \\
&\overset{(c)}{=} 3(1+\epsilon)\tilde{C}_j,
\end{aligned}
\tag{6.23}
$$

where (a) follows from definitions. Inequality (b) follows from (6.22) when y_{jl} and x_{jl} is seen as probabilities. Equality (c) comes from (6.9g) in (LP1).

6.9.3 Proof of Lemma 24

It is easy to observe that for every job j, $\bar{C}_j^\lambda \le C_j(\lambda)/\lambda$. The reason is that $C_j(\lambda)$ is the time that λ fraction of job j is completed in $\mathcal{S}$; therefore, in the stretched schedule $\bar{\mathcal{S}}$ by factor $1/\lambda$, job j is completed by time $C_j(\lambda)/\lambda$. Hence, we have

$$
\begin{aligned}
\mathbb{E}\left[\bar{C}_j^\lambda\right] \le \mathbb{E}\left[C_j(\lambda)/\lambda\right] &\overset{(a)}{=} \int_0^1 \frac{C_j(\lambda)}{\lambda} \times 2\lambda \times d\lambda \\
&\overset{(b)}{\le} 6(1+\epsilon)\tilde{C}_j,
\end{aligned}
$$

where Equality (a) is by definition of expectation with respect to λ, with pdf $f(\lambda) = 2\lambda$, and Equality (b) is due to Lemma 23.

6.10 De-randomization

In this section, we discuss how to de-randomize the random choice of $\lambda \in (0, 1]$ in SynchPack-1, which was used to construct schedule $\bar{\mathcal{S}}$ from schedule $\mathcal{S}$.

Recall that from Definition 14, $C_j(\lambda)$, $0 < \lambda \le 1$, is the starting point of the earliest interval in which λ-fraction of job j has been completed in schedule $\mathcal{S}$, which means at least λ-fraction of each of its tasks has been completed. We first aim to show that we can find

$$\lambda^\star = \arg\min_{\lambda\in(0,1]} \sum_{j\in\mathcal{J}} w_j C_j(\lambda)/\lambda \tag{6.24}$$

in polynomial time. Note that using the greedy packing algorithm, we schedule task fractions preemptively to form schedule $\mathcal{S}$. It is easy to see that $C_j(\lambda)$ is a step function with at most $O(L)$ breakpoints, since $C_j(\lambda) = d_l$ for some l and can get at most L different values. Consequently, $F(\lambda) = \sum_{j\in\mathcal{J}} w_j C_j(\lambda)$ is a step function with at most $O(NL)$ breakpoints. Let B denote the set of breakpoints of $F(\lambda)$. Thus, $F(\lambda)/\lambda = \sum_{j\in\mathcal{J}} w_j C_j(\lambda)/\lambda$ is a non-increasing function in intervals $(b, b']$, for b, b' being consecutive points in set B. This implies that,

$$\min_{\lambda\in(0,1]} F(\lambda)/\lambda = \min_{\lambda\in(0,1]} \sum_{j\in\mathcal{J}} w_j C_j(\lambda)/\lambda = \min_{\lambda\in B} \sum_{j\in\mathcal{J}} w_j C_j(\lambda)/\lambda.$$

We then can conclude that we can find $\lambda^\star$ in polynomial time by checking values of function $F(\lambda)/\lambda$ in at most $O(NL)$ points of set B and pick the one which incurs the minimum value. Given that, we have

$$\begin{aligned}\sum_{j\in\mathcal{J}} w_j \bar{C}_j^{\lambda^\star} &\le \sum_{j\in\mathcal{J}} (1+\epsilon) w_j C_j(\lambda^\star)/\lambda^\star \\ &\overset{(a)}{\le} (1+\epsilon)\mathbb{E}\Big[\sum_{j\in\mathcal{J}} w_j C_j(\lambda)/\lambda\Big] \\ &= (1+\epsilon)\sum_{j\in\mathcal{J}} w_j \int_{\lambda=0}^{1} \frac{C_j(\lambda)}{\lambda} 2\lambda d\lambda \\ &\overset{(b)}{=} 6(1+\epsilon)\sum_{j\in\mathcal{J}} w_j \tilde{C}_j,\end{aligned} \tag{6.25}$$

where (a) follows from (6.24). Equality (b) is due to Lemma 23. By choosing $\lambda = \lambda^\star$ in SynchPack-1, we have a deterministic algorithm with performance guarantee of $(6+\epsilon) \times$ OPT. , as stated by the following proposition.

6.11 Proofs Related to **SynchPack-2**

6.11.1 Proof of Lemma 25

Consider an optimal solution to the non-preemptive task scheduling problem with packing and synchronization constraints. For each task, we set ${z_{kj}^{il}}^{\star} = 1$ for the machine i and interval l if that task (k, j) is processed on i and finishes before d_l, and 0 otherwise. The rest of argument is similar to the proof of Lemma 21.

6.11.2 Proof of corollary 5

Note that (LP2) includes all the Constraints (6.9f)–(6.9h) of (LP1). Let α_l be the fraction of job j that is completed by interval l. Therefore,

$$\alpha_l = \sum_{l'=0}^{l} \tilde{x}_{jl'}. \tag{6.26}$$

Similar to Equations (6.23), we can write

$$\int_0^1 \hat{C}_j(\alpha)d\alpha = \sum_{l=0}^{L}(\alpha_l - \alpha_{l-1}) \times d_{l-1} = \sum_{l=0}^{L} \tilde{x}_{jl} d_{l-1} = \tilde{C}_j,$$

6.11.3 Proof of Lemma 26

We use the following fundamental theorem (Theorem 2.1.3 in [177]): If there exists a fractional matching of some value v in a bipartite graph G, then there exists an integral matching of the same value v in G on the non-zero edges and can be found in polynomial time.

In our constructed bipartite graph $\mathcal{G}$, edge weights w_{kj}^{ilc} can be seen as a fractional matching. This is because for any node $u \in U$, the sum of weights of edges that are incident to u is 1, while for any node $v \in V$ the sum of weights of edges that are incident to v is at most 1. Recall that $|\cup_{j\in\mathcal{J}} \mathcal{K}_j| = \sum_{j\in\mathcal{J}} \sum_{k\in\mathcal{K}_j} \sum_{l=0}^{L} \bar{z}_{kj}^{il}$ is the number of total tasks. Setting $G = \mathcal{G}$ and $v = |\cup_{j\in\mathcal{J}} \mathcal{K}_j|$, an integral matching of nodes in U to nodes in V on non-zero edges can be found in polynomial time by the stated theorem.

6.11.4 Proof of Lemma 27

We now present the proof of Lemma 27 which bounds $\mathcal{V}_{il}$ (the total volume of tasks matched to all copies of interval l for machine i) by the product of the capacity of machine i and the length of interval l. Observe that due to definition of v^i_{kj} and Constraint (6.9d) we have,

$$\sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}_j} v^i_{kj}\bar{z}^{il}_{kj} \le \bar{d}_l m_i, \tag{6.27}$$

The proof idea is similar to [172] that uses a simpler version of the mapping procedure in makespan minimization problem for scheduling tasks with unit resource requirements on unrelated machines with unit capacities, where each task can be scheduled in any machine. Let $\mathcal{V}^c_{il}$ denote the volume of the task that is matched to copy c of interval l on machine i. Thus, $\mathcal{V}_{il}$ is equal to the sum of $\mathcal{V}^c_{il}$ for all copies. Recall that we have $\lceil \bar{z}^{il}\rceil = \lceil \sum_{j\in J}\sum_{k\in\mathcal{K}_j}\bar{z}^{il}_{kj}\rceil$ many copies of interval l. Let $\mathcal{V}^{\max}_{il}$ denote the largest volume of the task that is mapped to interval l. For this task, we know that $\bar{z}^{il}_{kj} > 0$ because the integral matching was found on nonzero edges (line 23 in Algorithm 13); hence, $p^i_{kj} \le d_l = \lambda\bar{d}_l \le \bar{d}_l$ by Constraint (6.13b) and $\lambda \in (0,1]$. In addition, let $v^{\min_c}_{il}$ denote the volume of the smallest task that has an edge with non-zero weight to copy c of interval l in graph $\mathcal{G}$ (or equivalently, has a non-zero edge in the fractional matching.). Observe that, the volume of the task that is matched to copy $c+1$ is at most $v^{\min_c}_{il}$. This is because of the way we construct graph $\mathcal{G}$ by sorting tasks according to their volumes for each machine (see the ordering in (6.16)) and the way we assign weights to edges. Thus,

$$\begin{aligned}\mathcal{V}_{il} &= \sum_{c=1}^{\lceil \bar{z}^{il}\rceil}\mathcal{V}^c_{il} \le \mathcal{V}^{\max}_{il} + \sum_{c=2}^{\lceil \bar{z}^{il}\rceil} v^{\min_{c-1}}_{il} \\ &\overset{(a)}{\le} \bar{d}_l m_i + \sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}_j}\sum_{c=1}^{\lceil \bar{z}^{il}\rceil-1} v^i_{kj}w^{ilc}_{kj}.\end{aligned}$$

Inequality (a) comes from the fact that $\sum_{j\in J}\sum_{k\in\mathcal{K}_j} w_{kj}^{ilc} \leq 1$ and convex combination of some numbers is greater than the minimum number among them (note that the only copy for which we might have $\sum_{j\in J}\sum_{k\in\mathcal{K}_j} w_{kj}^{ilc} < 1$ is the last copy which is not considered in the left hand side of Inequality (a)). Therefore, as the direct result of the way we constructed graph $\mathcal{G}$, we have

$$\mathcal{V}_{il} \leq \bar{d}_l m_i + \sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}_j} v_{kj}^i \bar{z}_{kj}^{il}$$

6.11.5 Proof of Lemma 28

Lemma 28 ensures that we can accommodate all the task fractions mapped to machine-interval (i,l) within an interval with length twice $d_l + \sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}_j} v_{kj}^i \bar{z}_{kj}^{il}/m_i$.

Similar to Definition 13, we define $h(t)$ to be the height of the machine at time t. Assume that completion time of the last task, τ, is larger than $2V = 2\max(1,v)$, then

$$\sum_{j\in J} a_j p_j = \int_0^\tau h(t)dt > \int_0^{2V} h(t)dt \geq \int_0^V (h(t)+h(t+1))dt > 1+v,$$

where we have used the fact that $h(t)+h(t+1) > 1$, because otherwise the greedy scheduling can move tasks from time $t+1$ to time t as the greedy scheduling is non-preemptive and $p_j \leq 1$ for all tasks. Hence we arrived at a contradiction and the statement of Lemma 28 indeed holds.

6.11.6 Proof of Lemma 29

Let $\max(1,v) = 1$. We show correctness of Lemma 29 by constructing an instance for which an interval of size at least $2-\zeta$ is needed to be able to schedule all the tasks for any $\zeta > 0$. Given a $\zeta > 0$, consider $n > \log_2(1/\zeta)+1$ tasks with processing times $1, 1/2, 1/4, \ldots, 1/2^{(n-1)}$ and size $1/2+\eta$, for some $\eta > 0$ which is specified shortly. Note that we cannot place more than one of such tasks at a time on the machine, and therefore we need an interval of length $1 + 1/2 + 1/4 + \cdots + 1/2^{(n-1)} = 2 - 1/2^{(n-1)} > 2-\zeta$ to schedule all the tasks. The total volume of tasks is equal to $(1/2+\eta)(2-1/2^{(n-1)})$ which is less than 1, by choosing $\eta \leq 1/(2^{(n+1)}-2)$.

Therefore, for any $\zeta > 0$, we can construct an example for which an interval of length at least $2 - \zeta$ is needed to schedule all the tasks.

6.11.7 Proof of Lemma 30

Let T_{il} denote the completion time of the last task of machine-interval (i, l), and $\tau_{il'}$ be the length of the time interval that SynchPack-2 uses to schedule tasks of machine-interval (i, l). Then,

$$\begin{aligned} C_{kj} \leq T_{il} = \sum_{l'=0}^{l} \tau_{il'} &\overset{(a)}{\leq} 2 \times \sum_{l'=0}^{l} \big(\bar{d}_{l'} + \sum_{j' \in \mathcal{J}} \sum_{k' \in \mathcal{K}_{j'}} v^{i}_{k'j'} \bar{z}^{il'}_{k'j'}/m_i\big) \\ &\overset{(b)}{\leq} 4\bar{d}_l + 2 \sum_{l'=0}^{l} \sum_{j' \in \mathcal{J}} \sum_{k' \in \mathcal{K}_{j'}} v^{i}_{k'j'} \bar{z}^{il'}_{k'j'}/m_i \overset{(c)}{\leq} 6\bar{d}_l. \end{aligned} \tag{6.28}$$

Inequality (a) is due to Lemma 27 and Lemma 28, while Inequality (b) is because $d_{l'-1} = d_{l'}/2$. Further, Inequality (c) is by Constraint (6.9d).

6.12 Proofs Related to SynchPack-3

This section is devoted to the proof of the Theorem 13. We first characterize the solution of the linear program (LP3).

Lemma 31. *Let $\tilde{C}_j$ be the optimal solution to (LP3) for completion time of job j, as in the ordering (6.21). For each machine i and each job j, $m_i\tilde{C}_j \geq \frac{1}{2}\sum_{k=1}^{j} v_{ik}$.*

Proof. Proof. Using Constraint (6.20b), for any machine $i \in \mathcal{M}$, we have

$$v_{ij} m_i \tilde{C}_j \geq v_{ij}^2 + \sum_{j' \in \mathcal{J}, j' \neq j} v_{ij} v_{ij'} \delta_{j'j}.$$

Hence, by defining $\delta_{kk} = 0$, it follows that

$$\sum_{k=1}^{j} v_{ik} m_i \tilde{C}_k \geq \frac{1}{2}\Big(2\sum_{k=1}^{j} v_{ik}^2 + \sum_{k=1}^{j}\sum_{k'=1}^{j} \big(v_{ik}v_{ik'}\delta_{k'k} + v_{ik}v_{ik'}\delta_{kk'}\big)\Big) \tag{6.29}$$

We simplify the right-hand side of (6.29), using Constraint (6.20d), combined with the following equality

$$\sum_{k=1}^{j} v_{ik}^2 + \sum_{k=1}^{j} \sum_{\substack{k'=1 \\ k'\neq k}}^{j} v_{ik} v_{ik'} = (\sum_{k=1}^{j} v_{ik})^2,$$

and get

$$\sum_{k=1}^{j} v_{ik} m_i \tilde{C}_k \geq \frac{1}{2} \sum_{k=1}^{j} (v_{ik})^2 + \frac{1}{2} (\sum_{k=1}^{j} v_{ik})^2 \geq \frac{1}{2} (\sum_{k=1}^{j} v_{ik})^2. \tag{6.30}$$

Given that $\tilde{C}_j \geq \tilde{C}_k$ for $1 \leq k \leq j$, we get the final result. □ □

Let $C_j^\star$ be the completion time of job j in an optimal schedule, and $\text{OPT} = \sum_{j=1}^{N} w_j C_j^\star$ be the optimal value of our job scheduling problem. The following lemma states that the optimal value of (LP3), i.e., $\sum_{j=1}^{N} w_j \tilde{C}_j$, is a lower bound on the optimal value OPT.

Lemma 32. $\sum_{j=1}^{N} w_j \tilde{C}_j \leq \sum_{j=1}^{N} w_j C_j^\star = OPT.$

Proof. Proof. Consider an optimal preemptive solution to the task scheduling problem with packing and synchronization constraints. We set the ordering variables such that $\delta_{jj'} = 1$ if job j precedes job j' in this solution, and $\delta_{jj'} = 0$, otherwise. We note that this set of ordering variables and job completion times satisfies Constraint (6.20b) since this solution will respect resource constraints on the machines. It also satisfies Constraint (6.20c). Therefore, the optimal solution can be converted to a feasible solution to (LP1). This implies the desired inequality. □ □

Let C_{ij} and C_j denote the completion time of task (i, j) and the completion time of job j under SynchPack-3, respectively. In the next step for the proof of Theorem 13, we aim to bound the total volume of the first j jobs (according to ordering (6.21)) that are processed during the time interval $(0, 4\tilde{C}_j]$ and subsequently use this result to bound C_j. *Note that the list scheduling policy used in SynchPack-3 is similar to the one used in SynchPack-1*, without the extra consideration for placement of fractions corresponding to the same task on different machines. Thus, The arguments here are similar to the ones in Lemma 22. Nevertheless, we present them for completeness.

Let T_{ij} denote the first time that all the first j tasks complete under SynchPack-3 on machine i. Recall that, as a result of Constraint (6.20c) and ordering in (6.21), $\tilde{C}_j \geq \tilde{C}_k \geq p_{ik}$ for all $k \leq j$ and all $i \in \mathcal{M}$. Further, the height of machine i at time t restricted to the first j jobs is denoted by $h_{ij}(t)$ and defined as the height of machine i at time t when only considering the first j jobs according to the ordering (6.21). We have the following lemma.

Lemma 33. *Consider any interval $(\mathcal{T}_1, \mathcal{T}_2]$ for which $\mathcal{T}_2 - \mathcal{T}_1 = 2\tilde{C}_j$ and suppose $\mathcal{T}_2 < T_{ij}$ for some machine i. Then*

$$\sum_{t=\mathcal{T}_1+1}^{\mathcal{T}_2} h_{ij}(t) > m_i \tilde{C}_j \tag{6.31}$$

□

Proof. Proof of Lemma 33. Without loss of generality, consider interval $(0, 2\tilde{C}_j]$ and assume $T_{ij} > 2\tilde{C}_j$. Let $S_{ij}(\tau)$ denote the set of tasks (i,k), $k \leq j$ (according to ordering (6.21)), running at time τ on machine i. We construct a bipartite graph $G = (U \cup V, E)$ as follows. With a slight abuse of notations, for each time slot $\tau \in \{1, \ldots, 2\tilde{C}_j\}$ we consider a node τ, and define $U = \{\tau | 1 \leq \tau \leq \tilde{C}_j\}$, and $V = \{\tau | \tilde{C}_j + 1 \leq \tau \leq 2\tilde{C}_j\}$. For any $s \in U$ and $t \in V$, we add an edge (s,t) if $S_{ij}(t) \setminus S_{ij}(s) \neq \emptyset$, i.e., there is a task (i,k), $k \leq j$, running at time t that is not running at time s. Note that existence of edge (s,t) implies that $h_{ij}(s) + h_{ij}(t) > m_i$, because otherwise SynchPack-3 would have scheduled the task(s) in $S_{ij}(t) \setminus S_{ij}(s)$ (those that are running at t but not at s) at time s.

Next, we show that a perfect matching of nodes in U to nodes in V always exists in G. The existence of perfect matching then implies that any time slot $s \in (0, \tilde{C}_j]$ can be matched to a time slot $t \in (\tilde{C}_j, 2\tilde{C}_j]$ (one to one matching) and $h_{ij}(s) + h_{ij}(t) > m_i$. To prove that such a perfect matching always exists, we use Hall's Theorem [175]. For any set of nodes $\tilde{U} \subseteq U$, we define set of its neighbor nodes as $N_{\tilde{U}} = \{t \in V | \exists\, s \in \tilde{U} : (s,t) \in E\}$. Hall's Theorem states that a perfect matching exists if and only if for any $\tilde{U} \subseteq U$ we have $|\tilde{U}| \leq |N_{\tilde{U}}|$, where $|\cdot|$ denotes set cardinality (size). To arrive at a contradiction, suppose there is a (non-empty) set of nodes $\tilde{U} \subseteq U$ such that $|\tilde{U}| > |N_{\tilde{U}}|$. This implies that for a node t_1 in V but not in the neighbor set of $\tilde{U}$, i.e., $t_1 \in V \setminus N_{\tilde{U}}$, we should have

$$S_{ij}(t_1) \setminus S_{ij}(s) = \emptyset, \tag{6.32}$$

for all $s, s \in \tilde{U}$. We now consider two cases:

Case (i): $|V \setminus N_{\tilde{U}}| = 1$, which means $|N_{\tilde{U}}| = \tilde{C}_j - 1$. But we had assumed $|\tilde{U}| > |N_{\tilde{U}}|$, thus $|\tilde{U}| = \tilde{C}_j$ and $\tilde{U} = U$. This implies that the tasks that are running at time t_1, are also running in the entire interval $(0, \tilde{C}_j]$; therefore, the processing time of each of them is at least $\tilde{C}_j + 1$ which contradicts the fact that $\tilde{C}_j \geq p_{ik}$ for all jobs $k \leq j$, by Constraint (6.20c) and ordering in (6.21).

Case (ii): $|V \setminus N_{\tilde{U}}| > 1$. In addition to the previous node t_1, consider another node $t_2 \in V \setminus N_{\tilde{U}}$, and without loss of generality, assume $t_1 < t_2$. Similarly to (6.32), it holds that

$$S_{ij}(t_2) \setminus S_{ij}(s) = \emptyset, \tag{6.33}$$

for all $s \in \tilde{U}$. We claim that $S_{ij}(t_2) \subseteq S_{ij}(t_1)$, otherwise SynchPack-3 would have moved some task (i, k) running at t_2 and not at t_1 to time t_1 without violating machine i's capacity. This is feasible because, in view of (6.32) and (6.33), $(S_{ij}(t_1) \cup (i, k)) \setminus S_{ij}(s) = \emptyset$ for all $s \in \tilde{U}$. This implies that SynchPack-3 has scheduled all tasks of the set $S_{ij}(t_1) \cup (i, k)$ simultaneously at some time slot $s \in (0, \tilde{C}_j]$, which in turn implies that adding task (i, k) to time t_1 is indeed feasible (the total resource requirement of the tasks won't exceed m_i). Repeating the same argument for the sequence of nodes $t_1, t_2, \ldots, t_{|V \setminus N_{\tilde{U}}|}$, where $t_1 < t_2 < \cdots < t_{|V \setminus N_{\tilde{U}}|}$, we conclude that there exists a task that is running at all the times $t, t \in V \setminus N_{\tilde{U}}$, and at all the times $s \in \tilde{U}$. Therefore, its processing time is at least $\tilde{C}_j - |N_{\tilde{U}}| + |\tilde{U}|$ which is greater than $\tilde{C}_j$ by our assumption of $|\tilde{U}| > |N_{\tilde{U}}|$. This is a contradiction with the fact that $p_{ik} \leq \tilde{C}_j$ for all $k \leq j$ by Constraint (6.20c) and ordering (6.21).

Hence, we conclude that conditions of Hall's Theorem hold and a perfect matching in the constructed graph exists. As we argued, if $s \in U$ is matched to $t \in V$, we have $h_{ij}(s) + h_{ij}(t) > m_i$. Hence it follows that $\sum_{t=1}^{2\tilde{C}_j} h_{ij}(t) > m_i \tilde{C}_j$. □

□

Now we are ready to complete the proof of Theorem 13 regarding the performance of SynchPack-

3.

Proof. Proof of Theorem 13. Recall that C_{ij} and C_j denote completion time of task (i, j) and completion time of job j under SynchPack-3, respectively. Also, T_{ij} denotes the first time that all the first j tasks are completed under SynchPack-3 on machine i. Therefore, $C_{ij} \leq T_{ij}$, by definition.

Define i_j to be the machine for which $C_j = C_{i_j j}$. If $T_{ij} \leq 4\tilde{C}_j$ for all machines $i \in \mathcal{M}$ and all jobs $j \in \mathcal{J}$, we can then argue that $\sum_{j=1}^{N} w_j C_j \leq 4 \times \text{OPT}$, because

$$\sum_{j=1}^{N} w_j C_j = \sum_{j=1}^{N} w_j C_{i_j j} \leq \sum_{j=1}^{N} w_j T_{i_j j} \overset{(a)}{\leq} 4 \sum_{j=1}^{N} w_j \tilde{C}_j \overset{(b)}{\leq} 4 \sum_{j=1}^{N} w_j C_j^{\star},$$

where Inequality (a) follows from our assumption that $T_{ij} \leq 4\tilde{C}_j$, and Inequality (b) follows from Lemma 32.

Now to arrive at a contradiction, suppose $T_{ij} > 4\tilde{C}_j$ for some machine i and job j. We then have,

$$\begin{aligned} \sum_{k=1}^{j} v_{ik} = \sum_{t=1}^{T_{ij}} h_{ij}(t) &\overset{(c)}{>} \sum_{t=1}^{2\tilde{C}_j} h_{ij}(t) + \sum_{t=1}^{2\tilde{C}_j} h_{ij}(t + 2\tilde{C}_j) \\ &\overset{(d)}{>} m_i\tilde{C}_j + m_i\tilde{C}_j = 2m_i\tilde{C}_j, \end{aligned} \tag{6.34}$$

where Inequality (c) is due to the assumption that $T_{ij} > 4\tilde{C}_j$, and Inequality (d) follows by applying Lemma 33 twice, once for interval $(0, 2\tilde{C}_j]$ and once for interval $(2\tilde{C}_j, 4\tilde{C}_j]$. But (6.34) contradicts Lemma 31. Hence, $\sum_{j=1}^{N} w_j C_j \leq 4 \times \text{OPT}$. □ □

6.13 Supplementary Material Related to Simulations

6.13.1 Data Set

The data set is from a large Google cluster [151]. The original trace is over a month long period. To keep things simpler, we extract multi-task jobs of production scheduling class that were completed without any interruptions.In our experiments, we filter jobs and consider those with at most 200 tasks, which constitute about 99% of all the jobs in the production class. Also, in order to have reasonable traffic density on each machine (since otherwise the problem is trivial), we consider a cluster with 200 machines and randomly map machines of the original set to machines of this set. The final data set used for our simulations contains 7521 jobs with an average of 10 tasks per job. We also extracted memory requirement of each task and its corresponding processing time from the data set. In the data set, each job has a priority that represents its sensitivity to latency. There are 9 different values of job priorities.

We evaluate the performance of algorithms in both offline and online settings. For the offline setting, we consider the first 1000 jobs in the data set and assume all of these jobs are in the system at time 0. For the online setting, all the 7521 jobs arrive according to the arrival times information in the data set. Further, we consider 3 different cases for weight assignments: 1) All jobs have equal weights, 2) Jobs are assigned random weights between 0 and 1, and 3) Jobs' weights are determined based on the job priority and class information in the data set.

6.13.2 Algorithms

1. PSRS [169]: *Preemptive Smith Ratio Scheduling* is a preemptive algorithm for the parallel task scheduling problem (see Section 6.1.1) on a single machine. Modified Smith ratio of task (i, j) is defined as $\frac{w_j}{a_{ij}p_{ij}} = \frac{w_j}{v_{ij}}$. Moreover, a constant $\nu = 0.836$ is used in the algorithm. It also defines $T(a, t)$ to be the first time after t at which at least a amount of the machine's capacity is available, given the schedule at time t. On machine i, the algorithm first orders tasks based on the modified Smith ratio (largest ratio first). It then removes the first task (i, j) in the list and as long as the task needs at most 50% of the machine capacity m_i, it schedules the task in a non-preemptive fashion at the first time that available capacity of the machine is equal to or greater than the task's size, namely at $T(a_{ij}, t)$ where t is the current time and a_{ij} is the size of task (i, j). However, if task (i, j) requires more than half of the machine's capacity, the algorithm determines the difference $T(a_{ij}, t) - T(m_i/2, t)$. If this time difference is less than the ratio p_{ij}/ν, it schedules task (i, j) in the same way as those tasks with smaller size; that is, (i, j) starts at $T(a_{ij}, t)$ and runs to completion.

Otherwise at time $T(m_i/2, t) + p_{ij}/v$, it preempts all the tasks that do not finish before that time, and starts task (i, j). After task (i, j) is completed, those preempted tasks are resumed.

Recall that N is the number of jobs and M is the number of machines in the system. The time complexity of PSRS is $O(N^2)$ on each machine, as there are at most N tasks on each machine and there is at most N preemptions for each of them. Considering all the machines, the time complexity of PSRS is $O(MN^2)$.

For the online setting, upon arrival of each task, the algorithm preempts the schedule, updates the list, and schedule the tasks in a similar fashion.

2. Tetris [145]: Tetris is a heuristic that schedules tasks on each machine according to an ordering based on their scores (Section 6.1.1). Tetris was originally designed for the case that all jobs have identical weights; therefore, we generalize it by incorporating weights in tasks' scores. For each task (i, j) at time t, its score is defined as $s_{ij} = w_j(a_{ij} + \frac{\epsilon}{\sum_i a_{ij} p^t_{ij}})$, where $\epsilon = \frac{\sum_i \sum_j w_j a_{ij}}{\sum_j w_j (\sum_i a_{ij} p^t_{ij})^{-1}}$, and p^t_{ij} is the task's remaining processing time at time t. Note that the first term in the score depends on the task' size (it favors a larger task if it fits in the machine's remaining capacity), while the second term prefers a task whose job's remaining volume (based on the sum of its remaining tasks) is smaller. On each machine, Tetris orders tasks based on their scores and greedily schedules tasks according to the list as far as the machine capacity allows. We consider two versions of Tetris, preemptive (Tetris-p) and non-preemptive (Tetris-np).

In Tetris-p, upon completion of a task (or arrival of a job, in the online setting), it preempts the schedule, update the list, calculate scores based on updated values, and schedule the tasks in a similar fashion. The time complexity of Tetris-p is $O(N^2 \log(N))$ on each machine, as there are at most N preemptions, and at each preemption the algorithms needs to calculate and sort the scores. The total complexity is then $O(MN^2 \log(N))$ considering all the machines.

In Tetris-np, the algorithm does not preempt the tasks that are running; however, calculates scores for the remaining tasks based on updated values. Recall that we denote the maximum number of tasks a job has by K. For Tetris-np, the time complexity to calculate and sort the scores is $O(KNM \log(KNM))$ which should be done at most KN times. Therefore, the time complexity of Tetris-np is at most $O(K^2N^2M\log(KNM))$.

To take the placement constraint into account, Tetris imposes a remote penalty to the computed score to penalize use of remote resources. This remote penalty is suggested to be $\approx 10\%$ by [145]. In simulations, we also simulated Tetris by penalizing scores by the factor α, and found out that the performance is slightly better. Hence, we only report performance of Tetris with this remote penalty.

3. JSQ-MW [163]: Join-the-Shortest-Queue routing with Max Weight scheduling (JSQ-MW) is a non-preemptive algorithm in presence of data locality (Section 6.1.1). It assigns an arriving task to the shortest queue among those corresponding to the $\zeta = 3$ local servers with its input data and the remote queue. When a server is available, it either process a task from its local queue or from the remote queue, where the decision is made based on a MaxWeight scheme. We further combine JSQ-MW with the greedy packing scheme so it can pack and schedule tasks non-preemptively in each server.

To evaluate complexity of JSQ-MW, we note that in the routing step, we need to compare $\zeta + 1$ queue lengths for each task. Therefore, the complexity of this step is $O(\zeta KN)$. In the scheduling step, for each available machine, we need to compare its queue and the remote queue. Thus, the complexity of the scheduling step is $O(\zeta KN)$ as an availability of a machine is checked upon completion time of a task. Hence, the overall complexity is $O(KN)$ if ζ is constant, and $O(KNM)$ if $\zeta = \Omega(M)$.

4. SynchPack-2 and SynchPack-3: These are our non-preemptive and preemptive algorithms as described in Section 6.4 and Section 6.5. The complexity of our algorithms is mainly dominated by solving their corresponding LPs. While (LP3) has reasonable size and can be solved quickly (see Section 6.8 for the details), (LP2) requires more memory for large instances. In this case, to expedite computation, besides the 3 randomly chosen local machines that can schedule a task, we consider 10 other machines (5% of the machines, instead of all the machines) that can process the task in an α times larger processing time. We choose these 10 machines randomly as well. Note that this may degrade the performance of our algorithm, nevertheless, as will see, they still significantly outperform the past algorithms. See Section 6.8 for a copmrehensive discussion on time complexity of these algorithms.

A natural extension of our algorithms to online setting is as follows. We choose a parameter τ that is tunable. We divide time into time intervals of length τ. For the preemptive case, at the beginning of each interval, we preempt the schedule, update the processing times, and run the offline algorithm on a set of jobs, consisting of jobs that are not scheduled yet completely and those that arrived in the previous interval. In the non-preemptive case, tasks on the boundary of intervals are processed non-preemptively, i.e., we let the running tasks (according to the previously computed schedule) finish, then apply the non-preemptive offline algorithm on the updated list of jobs as in the preemptive online case, and proceed with the new schedule. Note that a larger value of τ reduces the complexity of the online algorithm; but it also decreases the overall performance. We use an adaptive choice of τ to improve the performance of our online algorithm, starting from smaller value of τ. In our simulations, we choose the length of the i-th interval, τ_i, as $\tau_i = \tau_0/(1 + \gamma \times \exp(-\beta i))$, $i = 1, 2, \cdots$, for some constants γ and β. We choose $\tau_0 = 3 \times 10^2$ seconds, which is 5 times greater than the average inter-arrival time of jobs, and $\gamma = 50$ and $\beta = 3$.

6.14 Pseudocodes of $(6 + \epsilon)$-Approximation Algorithm

A pseudocode for our preemptive $(6 + \epsilon)$-approximation algorithm SynchPack-1 described in Section 6.3 is given in Algorithm 11. Line 1 in Algorithm 11 corresponds to Step 1 in Section 6.3, lines 2-18 correspond to Step 2, construction of schedule $\mathcal{S}$, and lines 19-20 describe Slow-Motion and construction of schedule $\bar{\mathcal{S}}$ in Step 3.

6.15 Pseudocode of 24-Approximation Algorithm

Algorithm 12 provides a pseudocode for our non-preemptive algorithm, SynchPack-2, described in Section 6.4. Line 1 in Algorithm 12 corresponds to Step 1 in SynchPack-2 and lines 2 corresponds to Step 2, namely, construction of preemptive schedule and applying Slow-Motion. Lines 3-11 describes the procedure of constructing a non-preemptive schedule using $\bar{\mathcal{S}}$ in Step 3.

Algorithm 13 describes the mapping procedure which is used as a subroutine in Algorithm 12.

6.16 Pseudocodes of (4)-Approximation Algorithm

Algorithm 14 provides a pseudocode for SynchPack-3, our preemptive 4-approximation algorithm, described in Section 6.5. The algorithm is a simple list scheduling based on the ordering obtained from (LP3).

Algorithm 11 Preemptive Scheduling Algorithm SynchPack-1

```
Given a set of machines $\mathcal{M} = \{1, ..., M\}$, a set of jobs $\mathcal{J} = \{1, ..., N\}$, and weights $w_j$, $j \in \mathcal{J}$:
1: Solve (LP1) and denote its optimal solution by $\{\tilde{z}_{kj}^{il};\ j \in \mathcal{J},\ k \in \mathcal{K}_j,\ i \in \mathcal{M},\ l \in \{0, 1, \dots, L\}\}$.
2: List non-zero task fractions (i.e., tasks $(k, j)$ with size $a_{ij}$ and non-zero fractional duration $\tilde{z}_{kj}^{il} p_{kj}^{i}$) such that
   task fraction $(k, j, i, l)$ appears before task fraction $(k', j', i', l')$, if $l < l'$. Task fractions within each interval and
   corresponding to different machines are ordered arbitrarily.
3: Let $Q$ be size of the list, i.e., the total number of task fractions in the list, and set $t = 0$.
4: while $Q > 0$, do
5:     For each machine $i \in \mathcal{M}$, set $h_i(t)$ to be the height of machine $i$ at $t$.
6:     Set $q = q' = 1$.
7:     while $q' \le Q$, do
8:         Denote the $q$-th task fraction in the list by $(k_q, j_q, i_q, l_q)$.
9:         if $h_{i_q}(t) + a_{k_q j_q} \le m_{i_q}$ and no fraction of task $(k_q, j_q)$ is running in any other machine at $t$, then
10:            Schedule task fraction $(k_q, j_q, i_q, l_q)$ to run on machine $i_q$ and remove it from the list.
11:            Update $h_{i_q}(t) \leftarrow h_{i_q}(t) + a_{k_q j_q}$.
12:        else
13:            Update $q \leftarrow q + 1$.
14:        end if
15:        Update $q' \leftarrow q' + 1$.
16:    end while
17:    Process the task fractions that were scheduled in line 9 and denote the first time a task fraction completes
   by $t'$.
18:    Let $l^\star$ be the corresponding interval of the first task fraction in the list, i.e., the interval with minimum value
   that has some unscheduled task fraction.
19:    for Each task fraction $(k, j, i, l)$ in the schedule, do
20:        Update $\tilde{z}_{kj}^{il} \leftarrow \tilde{z}_{kj}^{il} - (t' - t)/p_{kj}^{i}$, where $t' - t$ is the amount of time it gets processed.
21:        if $\tilde{z}_{kj}^{il} > 0$ and $l > l^\star$, then
22:            Add the task fraction $(k, j, i, l)$ back to the list such that it appears before task fraction $(k', j', i', l')$,
   if $l < l'$.
23:        end if
24:    end for
25:    Update the time $t \leftarrow t'$, and $Q$ to be size of the updated list.
26: end while
27: Denote the obtained schedule by $\mathcal{S}$. Choose $\lambda$ randomly from $(0, 1]$ with pdf $f(\lambda) = 2\lambda$.
28: Construct schedule $\bar{\mathcal{S}}$ by applying Slow-Motion with parameter $\lambda$ to $\mathcal{S}$. Process jobs according to $\bar{\mathcal{S}}$.
```

Algorithm 12 Non-Preemptive Scheduling Algorithm SynchPack-2

```
Given a set of machines $\mathcal{M} = \{1, ..., M\}$, a set of jobs $\mathcal{J} = \{1, ..., N\}$, and weights $w_j$, $j \in \mathcal{J}$:
 1: Solve (LP2) and denote its optimal solution by $\{\tilde{z}^{il}_{kj},\ j \in \mathcal{J},\ k \in \mathcal{K}_j,\ i \in \mathcal{M},\ 0 \le l \le L\}$.
 2: Apply Slow-Motion by choosing $\lambda$ randomly from $(0, 1]$ with pdf $f(\lambda) = 2\lambda$, and define $\bar{z}^{il}_{kj}$, as in (6.15).
 3: Run Algorithm 13 and output list of tasks that are mapped to each machine-interval $(i, l)$, $i \in \mathcal{M},\ l \le L$.
 4: for Each machine $i \in \mathcal{M}$, do
 5:     Set $t = 0$.
 6:     Set $h_i(0) = 0$ to be the height of machine $i$ at time 0.
 7:     for Each interval $l$, $0 \le l \le L$, do
 8:         List the task. Let $Q$ be the total number of tasks in the list.
 9:         while $Q > 0$, do
10:             Set $q = q' = 1$.
11:             while $q' \le Q$, do
12:                 Denote the $q$-th task in the list by $(k_q, j_q)$.
13:                 if $h_i(t) + a_{k_q j_q} \le m_i$, then
14:                     Schedule task $(k_q, j_q)$, remove it from the list, and update $h_i(t) \leftarrow h_i(t) + a_{ij_q}$.
15:                 else
16:                     Update $q \leftarrow q + 1$.
17:                 end if
18:                 Update $q' \leftarrow q' + 1$.
19:             end while
20:             Process the tasks that were scheduled in line 14 until a task some task completes and denote this
        time by $t'$.
21:             Update $p^i_{kj} \leftarrow p^i_{kj} - (t' - t)$ for the scheduled tasks.
22:             if $p^i_{kj} = 0$, then
23:                 Update $h_i(t) \leftarrow h_i(t) - a_{ij_q}$.
24:             end if
25:             Update the time $t \leftarrow t'$, and $Q$ to be size of the updated list.
26:         end while
27:     end for
28: end for
```

Algorithm 13 Procedure of Mapping Tasks to Intervals

```
Given a set of jobs $\mathcal{J} = \{1, ..., N\}$, with task volumes $v^i_{kj}$ on machine $i$, and values of $\bar{z}^{il}_{kj}$:
 1: Construct bipartite graph $\mathcal{G}_i = (U \cup V, E)$ as follows:
 2: For each task $(k, j)$, $j \in J$, $k \in \mathcal{K}_j$, add a node in $U$.
 3: for Each machine $i$, $i \in \mathcal{M}$, do
 4:     Order and re-index tasks such that: $v^i_{k_1 j_1} \geq v^i_{k_2 j_2} \geq \dots v^i_{k_{N_i} j_{N_i}} > 0$.
 5:     for Each interval $l$, $l \leq L$, do
 6:         Consider $\lceil \bar{z}^{il} \rceil = \lceil \sum_{j \in J} \sum_{k \in \mathcal{K}_j} \bar{z}^{il}_{kj} \rceil$ consecutive nodes in $V_i$, and set $W^{ic_l}_l = 0$ for $1 \leq c_l \leq \lceil \bar{z}^{il} \rceil$. Also
    set $c_l = 1$.
 7:         for $q = 1$ to $N_i$, do
 8:             $R = \bar{z}^{il}_{kj}$,
 9:             while $R \neq 0$, do
10:                 Add an edge between the node $(k_q, j_q)$ in set $U$ and node $c_l \in V_i$.
11:                 Assign weight $w^{ilc}_{kj} = \min\{R, 1 - W^{c_l}_l\}$.
12:                 Update $R \leftarrow R - w^{ilc}_{kj}$.
13:                 Update $W^{c_l}_l \leftarrow W^{c_l}_l + w^{ilc}_{kj}$
14:                 if $W^{c_l}_l = 1$, then
15:                     $c_l = c_l + 1$.
16:                 end if
17:             end while
18:         end for
19:     end for
20: end for
21: Set $V = \cup_{i \in \mathcal{M}} V_i$.
22: Find an integral matching in $\mathcal{G}$ on the nonzero edges with value $| \cup_{j \in \mathcal{J}} \mathcal{K}_j | = \sum_{j \in \mathcal{J}} \sum_{k \in \mathcal{K}_j} \sum_{l=0}^{L} \bar{z}^{il}_{kj}$.
```

Algorithm 14 Preemptive Scheduling Algorithm SynchPack-3

```
Given a set of machines $\mathcal{M} = \{1, ..., M\}$, a set of jobs $\mathcal{J} = \{1, ..., N\}$, and weights $w_j$, $j \in \mathcal{J}$:
 1: Solve (LP1) and denote its optimal solution by $\{\tilde{C}_j; j \in \mathcal{J}\}$.
 2: Order and re-index jobs such that $\tilde{C}_1 \leq \tilde{C}_2 \leq ... \leq \tilde{C}_N$.
 3: for Each machine $i \in \mathcal{M}$, do
 4:     List tasks of machine $i$ respecting the ordering in line 2. Let $Q$ be the size of the list and set $t = 0$.
 5:     while $Q > 0$, do
 6:         Set $h_i(t)$ to be the height of machine $i$ at $t$.
 7:         Set $q = q' = 1$.
 8:         while $q' \leq Q$, do
 9:             Denote the $q$-th task in the list by $(i, j_q)$
10:             if $h_i(t) + a_{ij_q} \leq m_i$, then
11:                 Schedule task $(i, j_q)$, remove it from the list, and update $h_i(t) \leftarrow h_i(t) + a_{ij_q}$.
12:             else
13:                 Update $q \leftarrow q + 1$.
14:             end if
15:             Update $q' \leftarrow q' + 1$.
16:         end while
17:         Process the tasks that were scheduled in line 11 until some task completes and denote this time by $t'$.
18:         Update $p_{ij} \leftarrow p_{ij} - (t' - t)$ for the scheduled tasks.
19:         if $p_{ij} > 0$ and $\exists$ a task with $j' < j$ in the list, then
20:             Add the task $(i, j)$ back to the list respecting the ordering in line 2
21:         end if
22:         Update the time $t \leftarrow t'$, and $Q$ to be size of the updated list.
23:     end while
24: end for
```

Chapter 7: Conclusion and Discussion

In this dissertation, we proposed various theoretically sound algorithms for solving different scheduling problems in large-scaled data centers. For each algorithm, we proved that it achieves a performance objective related to the problem model. For many cases, what we offered is the algorithm's performance in the worst case scenario. We also evaluated all the proposed algorithms through extensive simulations. Besides, for each algorithm, we studied the algorithm's time complexity or how much overhead it adds to the system. We showed that in many cases, the proposed algorithm is computationally efficient. In the rest of this section, we provide summary of our results and discuss some future research direction.

7.1 Summary of Results

In what follows, we summarize the contributions of each chapter followed by some practical considerations:

- **Chapter 2:** This chapter presented a simple myopic algorithm that dynamically adjusts the link weights as a function of the link congestions and places any newly generated flow on a least weight path in the network, with no splitting/migration of existing flows. We demonstrated both theoretically and experimentally that this myopic algorithm has a good load balancing performance. In particular, we proved that the algorithm asymptotically minimizes a network cost and established the relationship between the network cost and the corresponding weight construct. Although our theoretical result is an asymptotic result, our experimental results show that the algorithm in fact performs very well under a wide range of traffic conditions and different data center networks. While the algorithm has low complexity, the real implementation depends on how fast the weight updates and least weight paths can be computed in practical data centers (e.g., based on SDN). One possible way to improve the computation time-scale is to perform the computation periodically or only for long flows, while using the previously computed least weight paths for short flows or between the periodic updates. Another possibility is to use the randomized versions of our myopic algorithm with an optimized parameter k which only takes a small random subset of available paths into account and finds the shortest path among them. While this algorithm has much lower complexity, it performs very well in structured topologies such as FatTree for small k. Finally, we would like to note that our myopic algorithm and its randomized versions can be directly applied to scheduling flowlets instead of scheduling flows, which can give higher rate/granularity of flows [18, 48].

- **Chapter 3:** In this chapter, we studied the problem of scheduling of coflows with release dates to minimize their total weighted completion time, and proposed an algorithm with improved approximation ratio. This algorithm is currently the state-of-the-art approximation algorithm for coflow scheduling. We also conducted extensive experiments to evaluate the performance of our algorithm, compared with three algorithms proposed before, using both real and synthetic traffic traces. Our experimental results show that our algorithm in fact performs very close to optimal.

- **Chapter 4:** In this chapter, we proposed algorithms for scheduling coflows of multi-stage jobs in order to minimize their makespan or total weighted completion time. In particular, our algorithms for total weighted completion time minimization provide significant improvements over the past known result for this problem. Moreover, our simulation results based on real traffic traces show that indeed our algorithm improves the total jobs' completion times in practice as well.

- **Chapter 5:** In this chapter, we studied max-min fair scheduling of multi-task jobs. We showed that it is NP-hard to find a schedule in which a sublinear number of jobs conform to their optimal max-min fair solution. We further used this result to show that some other scheduling problems considered in the literature of distributed computing are NP-hard, that were not proved before. We then defined two notions of approximation and developed approximation algorithms, using dynamic programming and random perturbation of tasks' processing times, with provable guarantees under the two approximation notions. Our experimental results show that our algorithms in fact perform very well under real traffic, in terms of both fairness and average performance.

- **Chapter 6:** We studied the problem of scheduling jobs, each job with multiple resource constrained tasks, in a cluster of machines. We proposed the first constant-approximation algorithms for minimizing the total weighted completion time of such jobs. The model and analysis in our setting of tasks with packing, synchronization, and placement constraints are new. Note that the approximation results are upper bounds on the algorithms' performance, and in fact our simulation results showed that the approximation ratios are very close to 1 in practice. As we showed, applying our simple greedy packing to schedule tasks mapped to each interval in SynchPack-2, provides a tight bound on the total volume of tasks and its relation to the associated linear program. Therefore, we cannot improve the final result by replacing this step with more intelligent bin packing algorithms like BestFit [178]. Although, in practice, applying such bin packing schemes can give a better performance. Further, throughout the chapter we assumed that tasks' resource requirements and durations are known to the scheduler. This can be justified by existence of well-established techniques that can provide estimates of tasks' resource requirements and processing times to the scheduler, based on the history of prior runs for recurring jobs, using tasks' peak demands, or measuring statistics from the first few tasks in each job, see [145, 146, 179, 180].

7.2 Future Directions

We now briefly describe some topics for future research based on the open problems that were emerged in this dissertation or as a result of generalizing the models we used.

- **Chapter 2:** The theoretical analysis of the randomized versions of our myopic algorithm can be an interesting open problem for future work.

- **Chapter 3:** As future work, other realistic constraints such as deadline constraints need to be considered for coflow scheduling problem. Also, theoretical and experimental evaluation of the performance of the proposed online algorithm is left for future work. While we modeled the data center network as a giant non-blocking switch (thus focusing on rate allocation), the routing of coflows in the data center network is also of great importance for achieving the quality of service.

- **Chapter 4:** The problem of multi-stage job scheduling is practically well-motivated, involve new challenges, and deserves further study. A few future research problems in this regard are the following. *General DAGs:* As we showed through an example, it is not possible for an approximation algorithm to provide a solution that is within $o(\sqrt{\mu})$ of the two simple lower bounds. Still, an interesting open problem is to improve the approximation algorithms for makespan and total weighted completion time in this case. *De-randomization:* Our algorithm for single job makespan minimization for rooted tree involves a random component (random choices of delays in Step 1 in DMA-SRT). There exists well-established techniques to de-randomized these steps and convert the algorithms to deterministic ones. For instance, given a set of path jobs, one approach for selecting good delays is to frame the problem as a vector selection problem and then apply techniques developed in [115, 116, 104].

- **Chapter 5:** Our theoretical guarantees for approximation algorithms in this chapter were concerned with equal utility functions. The analysis for unequal utility functions can be an interesting topic for a future work. Also we assumed machines are homogeneous. Incorporating inhomogeneous machines can be another future research.

- **Chapter 6:** Improving the performance bound of 24 requires a more careful and possibly different analysis. Further improvement of the result is a great topic for a future work. Extension of our model to capture multi-dimensional task resource requirements and analysis of online algorithms for our problem are also interesting and challenging topics for future work.

References

[1] Kai Chen et al. “Survey on routing in data centers: insights and future directions”. In: *IEEE network* 25.4 (2011).

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. “A scalable, commodity data center network architecture”. In: *ACM SIGCOMM Computer Communication Review* 38.4 (2008), pp. 63–74.

[3] Arjun Singh et al. “Jupiter rising: A decade of clos topologies and centralized control in Google’s datacenter network”. In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 183–197.

[4] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM*. Vol. 51. 1. ACM, 2008, pp. 107–113.

[5] Michael Isard et al. “Dryad: distributed data-parallel programs from sequential building blocks”. In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[6] Michael Mitzenmacher. “The power of two choices in randomized load balancing”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2001), pp. 1094–1104.

[7] Mosharaf Chowdhury and Ion Stoica. “Coflow: A networking abstraction for cluster applications”. In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM. 2012, pp. 31–36.

[8] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. “Efficient coflow scheduling with Varys”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 443–454.

[9] Bingchuan Tian et al. “Scheduling Coflows of Multi-stage Jobs to Minimize the Total Weighted Job Completion Time”. In: *IEEE INFOCOM 2018*. IEEE. 2018, pp. 864–872.

[10] Mosharaf Chowdhury and Ion Stoica. “Efficient coflow scheduling without prior knowledge”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. 2015, pp. 393–406.

[11] Software Foundation Apache. *Apache Hadoop*. `http://hadoop.apache.org`. 2018.

[12] Software Foundation Apache. *Apache Spark*. `https://spark.apache.org/docs/latest/index.html`. 2018.

[13] Ganesh Ananthanarayanan et al. "Disk-locality in datacenter computing considered irrelevant." In: *HotOS*. Vol. 13. 2011, pp. 12–12.

[14] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* (2010).

[15] Theophilus Benson et al. "MicroTE: Fine grained traffic engineering for data centers". In: *Proceedings of the 7th Conference on Emerging Networking Experiments and Technologies*. ACM. 2011, p. 8.

[16] Mohammad Al-Fares et al. "Hedera: Dynamic Flow Scheduling for Data Center Networks." In: *NSDI*. Vol. 10. 2010, pp. 19–19.

[17] Costin Raiciu et al. "Improving datacenter performance and robustness with multipath TCP". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 266–277.

[18] Srikanth Kandula et al. "Dynamic load balancing without packet reordering". In: *ACM SIGCOMM Computer Communication Review* 37.2 (2007), pp. 51–62.

[19] Albert Greenberg et al. "VL2: A scalable and flexible data center network". In: *ACM SIGCOMM Computer Communication Review*. Vol. 39. 4. 2009, pp. 51–62.

[20] Chuanxiong Guo et al. "BCube: A high performance, server-centric network architecture for modular data centers". In: *ACM SIGCOMM Computer Communication Review* 39.4 (2009), pp. 63–74.

[21] Milan Bradonjić, Iraj Saniee, and Indra Widjaja. "Scaling of capacity and reliability in data center networks". In: *ACM SIGMETRICS Performance Evaluation Review* 42.2 (2014), pp. 46–48.

[22] Ankit Singla et al. "Jellyfish: Networking Data Centers Randomly." In: *NSDI*. Vol. 12. 2012, pp. 17–17.

[23] Srikanth Kandula et al. "The nature of data center traffic: Measurements & analysis". In: *Proceedings of the 9th ACM SIGCOMM Conference On Internet Measurement Conference*. 2009, pp. 202–208.

[24] Abhishek Dixit et al. "On the impact of packet spraying in data center networks". In: *Proceedings of IEEE, INFOCOM, 2013*, pp. 2130–2138.

[25] Shimon Even, Alon Itai, and Adi Shamir. "On the complexity of time table and multi-commodity flow problems". In: *16th Annual Symposium on Foundation of Computer Science*. IEEE. 1975, pp. 184–193.

[26] Geoffrey M Guisewite and Panos M Pardalos. “Minimum concave-cost network flow problems: Applications, complexity, and algorithms”. In: *Annals of Operations Research* 25.1 (1990), pp. 75–99.

[27] Yefim Dinitz, Naveen Garg, and Michel X Goemans. “On the single-source unsplittable flow problem”. In: *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*. IEEE. 1998, pp. 290–299.

[28] Radhika Niranjan Mysore et al. “Portland: A scalable fault-tolerant layer 2 data center network fabric”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 39. 4. 2009, pp. 39–50.

[29] Jiaxin Cao et al. “Per-packet load-balanced, low-latency routing for clos-based data center networks”. In: *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*. ACM. 2013, pp. 49–60.

[30] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. “Understanding network failures in data centers: Measurement, analysis, and implications”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. 2011, pp. 350–361.

[31] Marco Chiesa, Guy Kindler, and Michael Schapira. “Traffic engineering with equal-cost-multipath: An algorithmic perspective”. In: *IEEE/ACM Transactions on Networking* 25.2 (2017), pp. 779–792.

[32] Siddhartha Sen et al. “Scalable, optimal flow routing in datacenters via local link balancing”. In: *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*. 2013, pp. 151–162.

[33] Joe Wenjie Jiang et al. “Joint VM placement and routing for data center traffic engineering”. In: *Proceedings of IEEE, INFOCOM, 2012*, pp. 2876–2880.

[34] Keqiang He et al. “Presto: Edge-based load balancing for fast datacenter networks”. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 465–478.

[35] Robert Gallager. “A minimum delay routing algorithm using distributed computation”. In: *IEEE transactions on communications* 25.1 (1977), pp. 73–85.

[36] Nithin Michael and Ao Tang. “Halo: Hop-by-hop adaptive link-state optimal routing”. In: *IEEE/ACM Transactions on Networking (TON)* 23.6 (2015), pp. 1862–1875.

[37] Dahai Xu, Mung Chiang, and Jennifer Rexford. “Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering”. In: *IEEE/ACM Transactions on networking* 19.6 (2011), pp. 1717–1730.

[38] Robert W Rosenthal. “A class of games possessing pure-strategy Nash equilibria”. In: *International Journal of Game Theory* 2.1 (1973), pp. 65–67.

[39] Noam Nisan et al. *Algorithmic game theory*. Vol. 1. Cambridge University Press Cambridge, 2007.

[40] Tim Roughgarden. *Selfish routing and the price of anarchy*. Vol. 174. MIT press Cambridge, 2005.

[41] Peter Key, Laurent Massoulié, and Don Towsley. “Path selection and multipath congestion control”. In: *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE. 2007, pp. 143–151.

[42] John Glen Wardrop. “ROAD PAPER. SOME THEORETICAL ASPECTS OF ROAD TRAFFIC RESEARCH.” In: *Proceedings of the institution of civil engineers* 1.3 (1952), pp. 325–362.

[43] David Applegate and Edith Cohen. “Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs”. In: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM. 2003, pp. 313–324.

[44] Hao Wang et al. “COPE: traffic engineering in dynamic networks”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 36. 4. ACM. 2006, pp. 99–110.

[45] Marcin Bienkowski, Miroslaw Korzeniowski, and Harald Räcke. “A practical algorithm for constructing oblivious routing schemes”. In: *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2003, pp. 24–33.

[46] Nick McKeown et al. “OpenFlow: Enabling innovation in campus networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[47] Martin Casado et al. “Rethinking enterprise network control”. In: *IEEE/ACM Transactions on Networking (TON)* 17.4 (2009), pp. 1270–1283.

[48] Mohammad Alizadeh et al. “CONGA: Distributed Congestion-Aware Load Balancing for Datacenters”. In: *Proceedings of the 2014 ACM conference on SIGCOMM*. 2014, pp. 503–514.

[49] Mehrnoosh Shafiee and Javad Ghaderi. “A simple congestion-aware algorithm for load balancing in datacenter networks”. In: *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE. 2016, pp. 1–9.

[50] Mehrnoosh Shafiee and Javad Ghaderi. “A simple congestion-aware algorithm for load balancing in datacenter networks”. In: *IEEE/ACM Transactions on Networking* 25.6 (2017), pp. 3670–3682.

[51] Alexander L Stolyar. “An infinite server system with general packing constraints”. In: *Operations Research* 61.5 (2013), pp. 1200–1217.

[52] Marco Chiesa, Guy Kindler, and Michael Schapira. “Traffic Engineering with Equal-Cost-MultiPath: An Algorithmic Perspective”. In: *Proceedings of IEEE, INFOCOM, 2014*, pp. 1590–1598.

[53] Bernard Fortz and Mikkel Thorup. “Internet traffic engineering by optimizing OSPF weights”. In: *Proceeding of 19th annual joint conference of the IEEE computer and communications societies. INFOCOM 2000*. Vol. 2, pp. 519–528.

[54] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[55] Deniz Ersoz, Mazin S Yousif, and Chita R Das. “Characterizing network traffic in a cluster-based, multi-tier data center”. In: *ICDCS’07. 27th International Conference on Distributed Computing Systems, 2007*. IEEE, pp. 59–59.

[56] Michael Grant and Stephen Boyd. *CVX: Matlab Software for Disciplined Convex Programming, version 2.1*. `http://cvxr.com/cvx`. Mar. 2014.

[57] Josep Díaz, Maria J Serna, and Nicholas C Wormald. “Bounds on the bisection width for random d-regular graphs”. In: *Theoretical Computer Science* 382.2 (2007), pp. 120–130.

[58] Béla Bollobás. *Random graphs*. Springer, 1998.

[59] Alexander L Stolyar and Yuan Zhong. “Asymptotic optimality of a greedy randomized algorithm in a large-scale service system with general packing constraints”. In: *Queueing Systems* 79.2 (2015), pp. 117–143.

[60] Javad Ghaderi, Yuan Zhong, and R Srikant. “Asymptotic optimality of BestFit for stochastic bin packing”. In: *ACM SIGMETRICS Performance Evaluation Review* 42.2 (2014), pp. 64–66.

[61] Patrick Billingsley. *Convergence of probability measures*. 2nd. John Wiley & Sons, 1999.

[62] Stewart N Ethier and Thomas G Kurtz. *Markov processes: Characterization and convergence*. Vol. 282. John Wiley & Sons, 2009.

[63] Konstantin Shvachko et al. “The hadoop distributed file system”. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE. 2010, pp. 1–10.

[64] Dhruba Borthakur. “The hadoop distributed file system: Architecture and design”. In: *Hadoop Project Website* 11.2007 (2007), p. 21.

[65] Fahad R Dogar et al. “Decentralized task-aware scheduling for data center networks”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 431–442.

[66] NM Mosharaf Kabir Chowdhury. *Coflow: A networking abstraction for distributed data-parallel applications*. University of California, Berkeley, 2015.

[67] Yangming Zhao et al. “RAPIER: Integrating routing and scheduling for coflow-aware data center networks”. In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE. 2015, pp. 424–432.

[68] Mosharaf Chowdhury et al. “Near Optimal Coflow Scheduling in Networks”. In: *SPAA ’19*. ACM, 2019, pp. 123–134.

[69] Zhen Qiu, Cliff Stein, and Yuan Zhong. “Minimizing the total weighted completion time of coflows in datacenter networks”. In: *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM. 2015, pp. 294–303.

[70] 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*. ACM. 2016, pp. 239–240.

[71] Mehrnoosh Shafiee and Javad Ghaderi. “Scheduling coflows in datacenter networks: Improved bound for total weighted completion time”. In: *ACM SIGMETRICS Performance Evaluation Review* 45.1 (2017), pp. 29–30.

[72] Kevin Jurcik. “Open Shop Scheduling to Minimize Makespan”. In: *Department of Mathematical Sciences Lakehead University Thunder Bay, Ontario* (2009).

[73] Yoo-Ah Kim. “Data migration to minimize the total completion time”. In: *Journal of Algorithms* 55.1 (2005), pp. 42–57.

[74] Leslie A Hall, David B Shmoys, and Joel Wein. “Scheduling to minimize average completion time: Off-line and on-line algorithms”. In: *SODA*. Vol. 96. 1996, pp. 142–151.

[75] Monaldo Mastrolilli et al. “Minimizing the sum of weighted completion times in a concurrent open shop”. In: *Operations Research Letters* 38.5 (2010), pp. 390–395.

[76] Saba Ahmadi et al. “On Scheduling Coflows”. In: *International Conference on Integer Programming and Combinatorial Optimization*. Springer. 2017, pp. 13–24.

[77] Mehrnoosh Shafiee and Javad Ghaderi. “Brief Announcement: A New Improved Bound for Coflow Scheduling”. In: *Proceedings of the 29th ACM symposium on Parallelism in Algorithms and Architectures*. ACM. 2017.

[78] Mehrnoosh Shafiee and Javad Ghaderi. “An improved bound for minimizing the total weighted completion time of coflows in datacenters”. In: *IEEE/ACM Transactions on Networking (TON)* 26.4 (2018), pp. 1674–1687.

[79] Hans Kellerer, Thomas Tautenhahn, and Gerhard Woeginger. “Approximability and non-approximability results for minimizing total flow time on a single machine”. In: *SIAM Journal on Computing* 28.4 (1999), pp. 1155–1166.

[80] Michael Pinedo. *Scheduling*. Springer, 2015.

[81] Linus Schrage. “Letter to the editor—a proof of the optimality of the shortest remaining processing time discipline”. In: *Operations Research* 16.3 (1968), pp. 687–690.

[82] Sushant Sachdeva and Rishi Saket. “Optimal inapproximability for scheduling problems via structural hardness for hypergraph vertex cover”. In: *Computational Complexity (CCC), 2013 IEEE Conference on*. IEEE. 2013, pp. 219–229.

[83] CN Potts. “An algorithm for the single machine sequencing problem with precedence constraints”. In: *Combinatorial Optimization II*. Springer, 1980, pp. 78–87.

[84] Rajiv Gandhi et al. “Improved bounds for scheduling conflicting jobs with minsum criteria”. In: *ACM Transactions on Algorithms (TALG)* 4.1 (2008), p. 11.

[85] James Renegar. “A polynomial-time algorithm, based on Newton’s method, for linear programming”. In: *Mathematical Programming* 40.1-3 (1988), pp. 59–93.

[86] Zhen Qiu, Clifford Stein, and Yuan Zhong. “Experimental Analysis of Algorithms for Coflow Scheduling”. In: *International Symposium on Experimental Algorithms*. Springer. 2016, pp. 262–277.

[87] Rayadurgam Srikant. *The mathematics of Internet congestion control*. Springer Science & Business Media, 2012.

[88] Dritan Nace and Michal Pióro. “Max-min fairness and its applications to routing and load-balancing in communication networks: a tutorial”. In: *IEEE Communications Surveys & Tutorials* 10.4 (2008).

[89] Shouxi Luo et al. “Towards Practical and Near-optimal Coflow Scheduling for Data Center Networks”. In: (2016).

[90] *Apache Hadoop*. `http://hadoop.apache.org`. 2019.

[91] Bingchuan Tian et al. “Scheduling dependent coflows to minimize the total weighted job completion time in datacenters”. In: *Computer Networks* 158 (2019), pp. 193–205.

[92] *Apache Hive*. `https://hive.apache.org`. 2019.

[93] Saksham Agarwal et al. “Sincronia: near-optimal network design for coflows”. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM. 2018, pp. 16–29.

[94] Sungjin Im et al. “Matroid Coflow Scheduling.” In: *ICALP*. 2019.

[95] Hamidreza Jahanjou, Erez Kantor, and Rajmohan Rajaraman. “Asymptotically optimal approximation algorithms for coflow scheduling”. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 2017, pp. 45–54.

[96] Yang Liu et al. “Scheduling Dependent Coflows with Guaranteed Job Completion Time”. In: *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE. 2016, pp. 2109–2115.

[97] Maurice Queyranne and Andreas S Schulz. “Approximation bounds for a general class of precedence constrained parallel machine scheduling problems”. In: *SIAM Journal on Computing* 35.5 (2006), pp. 1241–1253.

[98] Shi Li. “Scheduling to minimize total weighted completion time via time-indexed linear programming relaxations”. In: *SIAM Journal on Computing* 0 (2020), FOCS17–409.

[99] Robert Grandl et al. “GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters”. In: *OSDI’16*. 2016, pp. 81–97.

[100] Yu-Kwong Kwok and Ishfaq Ahmad. “Static scheduling algorithms for allocating directed task graphs to multiprocessors”. In: *ACM Computing Surveys (CSUR)* 31.4 (1999), pp. 406–471.

[101] Ronald L. Graham. “Bounds on multiprocessing timing anomalies”. In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429.

[102] Yu-Kwong Kwok and Ishfaq Ahmad. “Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors”. In: *IEEE transactions on parallel and distributed systems* 7.5 (1996), pp. 506–521.

[103] Edward Grady Coffman and John L Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[104] David B Shmoys, Clifford Stein, and Joel Wein. “Improved approximation algorithms for shop scheduling problems”. In: *SIAM Journal on Computing* 23.3 (1994), pp. 617–632.

[105] Leslie Ann Goldberg et al. “Better approximation guarantees for job-shop scheduling”. In: *SIAM Journal on Discrete Mathematics* 14.1 (2001), pp. 67–92.

[106] Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. “Chernoff–Hoeffding bounds for applications with limited independence”. In: *SIAM Journal on Discrete Mathematics* 8.2 (1995), pp. 223–250.

[107] Mehrnoosh Shafiee and Javad Ghaderi. “Scheduling Coflows with Dependency Graph”. In: *arXiv preprint arXiv:2012.11702* (2020).

[108] Teofilo Gonzalez and Sartaj Sahni. “Flowshop and jobshop schedules: complexity and approximation”. In: *Operations research* 26.1 (1978), pp. 36–52.

[109] Eugene L Lawler et al. “Sequencing and scheduling: Algorithms and complexity”. In: *Handbooks in operations research and management science* 4 (1993), pp. 445–522.

[110] Michael R Garey, David S Johnson, and Ravi Sethi. “The complexity of flowshop and jobshop scheduling”. In: *Mathematics of operations research* 1.2 (1976), pp. 117–129.

[111] David P Williamson et al. “Short shop schedules”. In: *Operations Research* 45.2 (1997), pp. 288–294.

[112] Garrett Birkhoff. “Tres observaciones sobre el algebra lineal”. In: *Univ. Nac. Tucumán Rev. Ser. A* 5 (1946), pp. 147–151.

[113] Eugene L Lawler and Jacques Labetoulle. “On preemptive scheduling of unrelated parallel processors by linear programming”. In: *Journal of the ACM (JACM)* 25.4 (1978), pp. 612–619.

[114] Donald Ervin Knuth. *The art of computer programming*. Vol. 3. Pearson Education, 1997.

[115] Prabhakar Raghavan and Clark D Tompson. “Randomized rounding: a technique for provably good algorithms and algorithmic proofs”. In: *Combinatorica* 7.4 (1987), pp. 365–374.

[116] Prabhakar Raghavan. “Probabilistic construction of deterministic algorithms: approximating packing integer programs”. In: *Journal of Computer and System Sciences* 37.2 (1988), pp. 130–143.

[117] Michael B Cohen, Yin Tat Lee, and Zhao Song. “Solving linear programs in the current matrix multiplication time”. In: *Proceedings of the 51st annual ACM SIGACT symposium on theory of computing*. 2019, pp. 938–942.

[118] Jan van den Brand. “A deterministic linear program solver in current matrix multiplication time”. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 259–278.

[119] Virginia Vassilevska Williams. “Multiplying matrices faster than Coppersmith-Winograd”. In: *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. 2012, pp. 887–898.

[120] François Le Gall. “Powers of tensors and fast matrix multiplication”. In: *Proceedings of the 39th international symposium on symbolic and algebraic computation*. 2014, pp. 296–303.

[121] Tom Leighton, Bruce Maggs, and Satish Rao. “Universal packet routing algorithms”. In: *29th Annual Symposium on Foundations of Computer Science*. IEEE. 1988, pp. 256–269.

[122] Charles Reiss et al. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis”. In: *Proc. of ACM Symposium on Cloud Computing*. 2012, p. 7.

[123] Ali Ghodsi et al. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.” In: *Nsdi*. Vol. 11. 2011. 2011, pp. 24–24.

[124] Wei Wang, Ben Liang, and Baochun Li. “Multi-resource fair allocation in heterogeneous cloud computing systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 26.10 (2015), pp. 2822–2835.

[125] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. “Dynamic resource allocation for shared data centers using online measurements”. In: *International Workshop on Quality of Service*. Springer. 2003, pp. 381–398.

[126] *Hadoop Fair Scheduler*. `http://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`. 2018.

[127] *Hadoop Capacity Scheduler*. `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`. 2018.

[128] Luis Diego Briceno et al. “Time utility functions for modeling and evaluating resource allocations in a heterogeneous computing system”. In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. 2011, pp. 7–19.

[129] Zhe Huang et al. “Need for speed: Cora scheduler for optimizing completion-times in the cloud”. In: *IEEE INFOCOM*. 2015, pp. 891–899.

[130] Jeffrey Jaffe. “Bottleneck flow control”. In: *IEEE Transactions on Communications* 29.7 (1981), pp. 954–962.

[131] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*. Vol. 2. Prentice-Hall International New Jersey, 1992.

[132] Benjamin Avi-Itzhak and Hanoch Levy. “On measuring fairness in queues”. In: *Advances in applied probability* 36.3 (2004), pp. 919–936.

[133] Zhe Huang et al. “RUSH: A robust scheduler to manage uncertain completion-times in shared clouds”. In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2016, pp. 242–251.

[134] Li Chen et al. “Optimizing coflow completion times with utility max-min fairness”. In: *INFOCOM 2016*. 2016, pp. 1–9.

[135] David E Irwin, Laura E Grit, and Jeffrey S Chase. “Balancing risk and reward in a market-based task service”. In: *International Symposium on High-Performance Distributed Computing,* 2004, pp. 160–169.

[136] Matei Zaharia et al. “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling”. In: *Proceedings of the 5th European Conference on Computer Systems*. ACM. 2010, pp. 265–278.

[137] Stratos Dimopoulos, Chandra Krintz, and Rich Wolski. “Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics”. In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 233–244.

[138] Reza Ahmadi, Uttarayan Bagchi, and Thomas A Roemer. “Coordinated scheduling of customer orders for quick response”. In: *Naval Research Logistics (NRL)* 52.6 (2005), pp. 493–512.

[139] Naveen Garg, Amit Kumar, and Vinayaka Pandit. “Order scheduling models: Hardness and algorithms”. In: *Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*. Springer. 2007, pp. 96–107.

[140] Joseph Y-T Leung, Haibing Li, and Michael Pinedo. “Scheduling orders for multiple product types to minimize total weighted completion time”. In: *Discrete Applied Mathematics* 155.8 (2007), pp. 945–970.

[141] Daniel Pérez Palomar and Mung Chiang. “A tutorial on decomposition methods for network utility maximization”. In: *IEEE Journal on Selected Areas in Communications* 24.8 (2006), pp. 1439–1451.

[142] Po-Lung Yu. “Domination structures and nondominated solutions”. In: *Multiple-Criteria Decision Making*. Springer, 1985, pp. 163–214.

[143] James P Evans and Ralph E Steuer. “A revised simplex method for linear multiple objective programs”. In: *Mathematical Programming* 5.1 (1973), pp. 54–72.

[144] Hanif D Sherali. “Equivalent weights for lexicographic multi-objective programs: Characterizations and computations”. In: *European Journal of Operational Research* 11.4 (1982), pp. 367–379.

[145] Robert Grandl et al. “Multi-resource packing for cluster schedulers”. In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 455–466.

[146] Sameer Agarwal et al. “Re-optimizing data-parallel computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. 2012, pp. 21–21.

[147] Mehrnoosh Shafiee and Javad Ghaderi. “On Max-Min Fairness of Completion Times for Multi-Task Job Scheduling”. In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 100–108.

[148] Matthias Ehrgott. *Multicriteria optimization*. Vol. 491. Springer Science & Business Media, 2005.

[149] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. W. H. Freeman New York, 2002.

[150] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[151] John Wilkes. *More Google cluster data*. Google research blog. Posted at `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`. Nov. 2011.

[152] Matei Zaharia et al. “Apache spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[153] Thomas Cheatham et al. “Bulk synchronous parallel computing–a paradigm for transportable software”. In: *Tools and Environments for Parallel and Distributed Systems*. Springer, 1996, pp. 61–76.

[154] Matei Zaharia et al. “Improving MapReduce performance in heterogeneous environments”. In: *Osdi*. Vol. 8. 4. 2008, p. 7.

[155] Ganesh Ananthanarayanan et al. “Reining in the Outliers in Map-Reduce Clusters using Mantri.” In: *Osdi*. Vol. 10. 1. 2010, p. 24.

[156] Karthik Kambatla et al. “Asynchronous algorithms in MapReduce”. In: *2010 IEEE International Conference on Cluster Computing (CLUSTER)*. 2010, pp. 245–254.

[157] Vlad Nitu et al. “Working Set Size Estimation Techniques in Virtualized Environments: One Size Does not Fit All”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.1 (2018), p. 19.

[158] Jeff Rasley et al. “Efficient queue management for cluster scheduling”. In: *Proceedings of the 11th European Conference on Computer Systems*. 2016, p. 36.

[159] Malte Schwarzkopf et al. “Omega: flexible, scalable schedulers for large compute clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 351–364.

[160] Jiahui Jin et al. “Bar: An efficient data locality driven task scheduling algorithm for cloud computing”. In: *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE. 2011, pp. 295–304.

[161] Abhishek Verma et al. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 18.

[162] Jinwei Liu and Haiying Shen. “Dependency-aware and resource-efficient scheduling for heterogeneous jobs in clouds”. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2016, pp. 110–117.

[163] Weina Wang et al. “Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality”. In: *IEEE/ACM Transactions on Networking (TON)* 24.1 (2016), pp. 190–203.

[164] Ali Yekkehkhany, Avesta Hojjati, and Mohammad H Hajiesmaili. “GB-PANDAS:: Throughput and heavy-traffic optimality analysis for affinity scheduling”. In: *ACM SIGMETRICS Performance Evaluation Review* 45.3 (2018), pp. 2–14.

[165] Zhi-Long Chen and Nicholas G Hall. “Supply chain scheduling: Conflict and cooperation in assembly systems”. In: *Operations Research* 55.6 (2007), pp. 1072–1089.

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

[167] Michael R Garey and Ronald L. Graham. “Bounds for multiprocessor scheduling with resource constraints”. In: *SIAM Journal on Computing* 4.2 (1975), pp. 187–200.

[168] Jacek Blazewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. “Scheduling subject to resource constraints: classification and complexity”. In: *Discrete applied mathematics* 5.1 (1983), pp. 11–24.

[169] Uwe Schwiegelshohn. “Preemptive weighted completion time scheduling of parallel jobs”. In: *SIAM Journal on Computing* 33.6 (2004), pp. 1280–1308.

[170] Jan Remy. “Resource constrained scheduling on multiple machines”. In: *Information Processing Letters* 91.4 (2004), pp. 177–182.

[171] Mehrnoosh Shafiee and Javad Ghaderi. “Scheduling Parallel-Task Jobs Subject to Packing and Placement Constraints”. In: *arXiv preprint arXiv:2004.00518* (2020).

[172] Jan Karel Lenstra, David B Shmoys, and Eva Tardos. “Approximation algorithms for scheduling unrelated parallel machines”. In: *Mathematical programming* 46.1-3 (1990), pp. 259–271.

[173] Maurice Queyranne and Maxim Sviridenko. “A (2+ epsilon)-approximation algorithm for the generalized preemptive open shop problem with minsum objective”. In: *Journal of Algorithms* 45.2 (2002), pp. 202–212.

[174] Andreas S Schulz and Martin Skutella. “Random-based scheduling new approximations and LP lower bounds”. In: *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer. 1997, pp. 119–133.

[175] Philip Hall. “On representatives of subsets”. In: *Journal of the London Mathematical Society* 1.1 (1935), pp. 26–30.

[176] Gurobi Optimization LLC. *Gurobi Optimizer*. 2018.

[177] Edward R Scheinerman and Daniel H Ullman. *Fractional graph theory: a rational approach to the theory of graphs*. Courier Corporation, 2011.

[178] Edward G Coffman Jr et al. “Performance bounds for level-oriented two-dimensional packing algorithms”. In: *SIAM Journal on Computing* 9.4 (1980), pp. 808–826.

[179] Kristi Morton, Magdalena Balazinska, and Dan Grossman. “ParaTimer: a progress indicator for MapReduce DAGs”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 507–518.

[180] Andrew D Ferguson et al. “Jockey: guaranteed job latency in data parallel clusters”. In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012, pp. 99–112.