Online Partial Throughput Maximization for Multidimensional Coflow

Sungjin Im*, Maryam Shadloo* and Zizhan Zheng†
* Electrical Engineering and Computer Engineering, University of California, Merced, CA 95343
Email: {sim3, mshadloo}@ucmerced.edu
† Dept. of Computer Science, Tulane University, New Orleans, LA 70118
Email: zzheng3@tulane.edu

***Abstract*—Coflow has recently been introduced to capture communication patterns that are widely observed in the cloud and massively parallel computing. Coflow consists of a number of flows that each represents data communication from one machine to another. A coflow is completed when all of its flows are completed. Due to its elegant abstraction of the complicated communication processes found in various parallel computing platforms, it has received significant attention.**

In this paper, we consider coflow for the objective of maximizing partial throughput. This objective seeks to measure the progress made for partially completed coflows before their deadline. Partially processed coflows still could be useful when their flows send out useful data that can be used for the next round computation. In our measure, a coflow is processed by a certain fraction when all of its flows are processed by the same fraction or more. We consider a natural class of greedy algorithms, which we call myopic concurrent. The algorithms seek to maximize the marginal increase of the partial throughput objective at each time. We analyze the performance of our algorithm against the optimal scheduler. In fact, our result is more general as a flow could be extended to demand various heterogeneous resources. Our experiment demonstrates our algorithm's superior performance.

I. Introduction

Coflow [1] has recently emerged as an elegant model that abstracts communication patterns that are frequently observed in cloud computing and massively parallel computing such as MapReduce [2] and Spark [3]. Each coflow consists of a number of parallel flows (data communications) and each flow has some data to transfer from one machine to another. A coflow is completed when all its flows are completed; for the formal model, see Section II-A. Due to its general model, coflow has received significant attentions since its introduction. Particularly, coflow has been studied for various objectives such as average completion time and throughput maximization.

In throughput maximization, each coflow typically has a deadline and gives a certain utility when completed before its deadline. However, in certain applications, partially processed coflows still could be useful when they contain useful data that can be used for the next round of computation. For example, suppose a coflow is intended to capture data migration to prepare for certain computation related to data analytics. To get the ideal analytics result, machines must receive the whole data set, but partial data can be used as samples for the large data set. As the machines receive more data over time, they produce more accurate outcomes.

The authors are ordered in alphabetical order.

Partial execution is becoming an important paradigm to support time-sensitive applications such as interactive services [4] and real-time data analytics [5], [6]. For many of these applications, a timely result with a slight loss in accuracy is preferable to the completed but delayed result. For example, given a web search query, the server could send out top ranked results followed by lower ranked results so that clients first see important results quickly and more results as time progresses.

Motivated by such scenarios, in this paper, we consider the objective of maximizing partial throughput, which was recently studied in [7], [8] in a cloud computing context. In their setting, each request/job consists of multiple identical tasks. Each task asks for multiple heterogeneous resources and tasks are processed when given the resources they demand. However, the tasks are homogeneous, and thus their setting is very different from coflow where each job (coflow) consists of communications (flows) between different machines. In coflow scheduling, each coflow (job) j is associated with a utility function f_j. The function takes as input how much each flow in j has been processed and outputs the associated coflow's utility. It is reasonable to assume that all flows of each coflow are equally important for the next round computation, and therefore, the usefulness of a partially processed coflow is bottlenecked by the least processed flow.

Further, we generalize coflow to the more general setting to capture heterogeneous resources. Coflow is certainly an elegant communication model that captures communication between machines that focus on each machine's network capacity, i.e., a maximum number of packets each machine can send or receive. However, there is a need to consider resources at the level of finer granules in certain settings. For example, one may want to consider computation and communication simultaneously if partial computation outcome can be used for the next round computation. In such cases, computing resources such as CPUs should be considered simultaneously together network bandwidth.

Indeed, multidimensional scheduling [9]–[13] has been extensively studied to capture jobs requiring heterogeneous resources — some jobs could be more CPU-intensive while others could be more memory-intensive. In multidimensional scheduling, there are multiple heterogeneous resources which are often assumed to be divisible. Then, each job demands certain resources for its execution and is processed at a certain rate that is determined by the resources it is allocated. The main motivation of this model was to study resource allocations in cloud computing, and therefore does not capture parallel communication between machines.

A. Our Scheduling Model

In this paper, we study a combination of the two aforementioned scheduling models. In the combined model, which we call *multidimensional coflow*, each job (or coflow) consists of tasks (flows), and each task requires certain resources *simultaneously*. Each task is processed at a rate depending on the least resource it is allocated relative to its demand for the resource. In other words, resources demanded by a task cannot be replaced by other resources. It is worth mentioning that our multidimensional coflow generalizes coflow, as well as multidimensional scheduling. Coflow is a special case of our setting where each task (flow) uses only two resources, namely ingress and egress ports it uses; for more details see II-A. Each job j has a utility function f_j that takes as input the fraction of job j processed and outputs the associated coflow's utility, which measures the importance of the partial result to the application. In general, the progress of a job depends on the progress of each task in the job. The relationship between the two can be modeled by a real valued function with multi-dimensional inputs and may be learned from history data. In this work, we make a first order approximation to this relationship and assume that job j has been processed by a fraction of z_j when all of its tasks have been processed by the same fraction. The goal is to maximize $\sum_j f_j(z_j)$ where z_j denotes the fraction of job j that has been processed before its deadline. We assume that f_j is monotone and concave for all j. A concave function captures the diminishing return property that is often observed in realty [4], [7]. For instance, a study of 200K queries in a production trace of Bing search engine shows that the relationship between the response quality and the amount of resources (or time) used is close to a monotone concave function [4]. Moreover, a concave utility can be used to achieve fairness between coflows. We seek to design an online scheduling algorithm that makes scheduling decisions without knowing jobs arriving in the future.

B. Our Contributions

Our contribution is largely two-fold. First, as mentioned above, our scheduling model naturally combines the two general and widely studied scheduling models, coflow and multidimensional scheduling. We believe multidimensional coflow can capture various scheduling environments and the model will find more applications in the future.

Second, we study multidimensional coflow for maximizing partial throughput online and find an effective scheduling algorithm. Our algorithm is very natural as at each time, it seeks to maximize the marginal increase of the objective. That is, if J_t denotes the set of currently alive jobs at time t and $z_{j,\leq t}$ denotes how much j has been processed by time t, the algorithm seeks to maximize $\sum_{j\in J_t}(f_j(z_{j,\leq t}) - f_j(z_{j,\leq t-1}))$. This algorithm has three interesting properties:

1) The algorithm is *online* as its scheduling decision does not depend on jobs that have not arrived yet.
2) The algorithm is *myopic* as it ignores how soon each job's deadline is but tries to maximize the marginal increase of the objective at each moment. In other words, the algorithm is oblivious to jobs' deadlines.
3) The algorithm yields a *concurrent* schedule in the sense that all tasks of the same job are processed by an equal fraction at any point in time.

It is worth emphasizing our algorithm is oblivious to jobs deadlines. Thus, even if the algorithm doesn't have accurate information about jobs deadlines, it can be readily implemented. Further, such algorithms are typically easier to implement as they do not have to track job deadlines or prioritize jobs based on their deadlines.

We study the performance of our *myopic concurrent* algorithm using the competitive analysis framework. That is, our algorithm's objective is compared against the offline optimal scheduler's objective and is said to be c-competitive if the former is at least c times the latter. Competitive analysis is of fundamental interest as it makes no stochastic assumptions on the input and gives a certain guarantee on the algorithm's performance even in the worst case [14].

Note that all tasks of the same job do not have to be processed simultaneously. As we mentioned, our algorithm, however, yields a concurrent schedule. Informally speaking, we show, if any schedule can be outperformed, in terms of partial throughput, by a concurrent schedule using ρ factor more resources, then our myopic concurrent algorithm is $1/(1+\rho)$-*competitive* against the optimal schedule that is not necessarily concurrent (Theorem 1). Since a concurrent schedule's objective could be only $1/\rho$ times the optimal schedule's objective, this implies our concurrent online algorithm's performance is pretty close to that of the best concurrent schedule that could be offline.

Then, we study an upper bound on the parameter ρ, which we define and call *price of concurrence* (Definition 2). We show the price of concurrence (PoC) is at most an easy-to-compute parameter (Theorem 2) that we call the maximum-to-average resource demand (MA, for short) parameter. Roughly speaking, the MA parameter is the worst maximum-to-average resource demand by any job.

From the analysis point of view, we give an intuitive analysis by making a connection between our problem and a submodular maximization problem. While such a connection was used in the previous work [15], our extension is more general since our problem itself, or even coflow, does not admit such a connection as it is. However, when the schedules are restricted to concurrent, an interpretation as submodular optimization becomes possible. Our clean analysis has a potential to find more applications and thus is of independent interest.

Finally, we show that for the partial throughput objective, our myopic concurrent algorithm considerably outperforms other existing algorithms developed for other objectives.

C. Organization

In Section II, we define our multidimensional coflow along with notations that will be used throughout the paper and discuss some of its applications. Further, some related work will be discussed. In Section IV, we present our myopic concurrent algorithm and state our results formally. Section V is devoted to proving our theoretical results. Then, we complement our theoretical results by a simulated experiment in Section VI and close with concluding remarks in Section VII.

II. Formal Problem Definition

We first give the formal definition of the multi-dimensional coflow problem (MD-Coflow), which we introduce and study

in this paper. In MD-Coflow, there is a set $\mathcal{R}$ of divisible resources that can be used to serve the clients. The set of resources remains the same throughout the whole time. By scaling, each resource is w.l.o.g. assumed to exist in one unit, meaning that one can use at most one unit of each resource at any point in time. Each resource has a specific type, and the set of resources of type k is denoted as $\mathcal{R}_k$. Let $|\mathcal{R}_k|$ be the number of resources of type k. Note that $|\mathcal{R}| = \sum_k |\mathcal{R}_k|$.

A set J of jobs arrive over time to be served. Each job j consists of a set of tasks, Q_j. All tasks of job j have the same arrival/release time a_j and desire to be completed by time b_j, which is referred to as the task's deadline, or equivalently, job j's deadline. We say that $[a_j, b_j]$ is j's lifespan. Each task $q \in Q_j$ has a demand d_{qr} for each resource $r \in \mathcal{R}$. Let $D_{qk} := \sum_{r \in \mathcal{R}_k} d_{qr}$ be the total demand of task q for all resources of type k. Similarly, let $D_{jk} := \sum_{q \in Q_j} D_{qk}$ denote the total demand of job j for all resources of type k.

To discuss how a task or job is processed, we need to fix the time model. In this paper, we assume a continuous time model where a scheduling decision is made at each instant of time. However, to make the presentation of our algorithm and analysis more transparent, we will often talk about time slots, which are assumed to be infinitesimally small.

A task q needs resources simultaneously in proportion to its demands to be processed. Formally, if x_{qrt} is the amount of resource r that task q receives at time t, task q is processed by a fraction of $\min_{r:d_{qr}\neq 0} x_{qrt}/d_{qr}$. Hence, we can assume w.l.o.g. that for every pair of task q and time t, x_{qrt}/d_{qr} is equal for all r such that $d_{qr} \neq 0$. Let y_{qt} be the fraction of task q that is processed at time t. We measure job j's processing by its least processed task. Formally, j is processed by $z_j = \min_{q \in Q_j} \sum_{t \in [a_j, b_j]} y_{qt}$. Each job j is associated with a utility function f_j, which is concave, monotone, and differentiable. Job j has a utility $f_j(z_j)$. The goal is to maximize total *partial throughput*, i.e., $\sum_j f_j(z_j)$. The objective is called partial objective as it measures the total utility of partially executed jobs before their deadline. A job can be processed by at most a fraction of 1. We can either explicitly add this constraint or prevent each function f_j from growing when z_j becomes greater than 1.

The following, $\mathsf{CP}_{\text{MD-Coflow}}$, is a convex programming formulation of MD-Coflow.

$$\max_{x,y,z} \sum_j f_j(z_j) \qquad \mathsf{CP}_{\text{MD-Coflow}}$$

$$\text{s.t.} \sum_{j,q \in Q_j} x_{qrt} \leq 1 \qquad \forall r \in \mathcal{R}, t \tag{1}$$

$$x_{qrt} \geq d_{qr} y_{qt} \qquad \forall r \in \mathcal{R}, j \in J, q \in Q_j, t \tag{2}$$

$$z_j \leq \sum_t y_{qt} \qquad \forall j \in J, q \in Q_j \tag{3}$$

$$x_{qrt} = 0 \qquad \forall r \in \mathcal{R}, j \in J, q \in Q_j, \forall t \notin [a_j, b_j] \tag{4}$$

$$x_{qrt} \geq 0 \qquad \forall r \in \mathcal{R}, j \in J, q \in Q_j, t$$

$$y_{qt} \geq 0 \qquad \forall j \in J, q \in Q_j, t$$

$$0 \leq z_j \leq 1 \qquad \forall j \in J$$

There are three variables in $\mathsf{CP}_{\text{MD-Coflow}}$: variable x_{qrt} denotes the amount of resource r that task q receives at time t; y_{qt} is the fraction of task q that is processed at time t; and z_j is the fraction of job j completed before j's deadline. Constraint (1) says that each resource r can be used by at most one unit at all times. Constraint (2) enforces the concurrent requirement at the level of tasks — task q's processing is constrained by the resource that q is given the least relative to its demand. In other words, task q is processed by $\min_{r:d_{qr}\neq 0} x_{qrt}/d_{qr}$ at time t. Constraint (3) sets z_j to $\min_{q \in Q_j} \sum_t y_{qt}$, meaning that j's processing is measured by the least processed task q among all of j's tasks. Constraint (4) ensures that jobs/tasks' deadlines are respected. Finally, a job is processed by at most a unit fraction. Note that we can assume w.l.o.g. that $\sum_t y_{qt} \leq 1$ for all tasks q.

In this paper, we seek to study online algorithms for MD-Coflow. An online algorithm becomes aware of a job j only when the job arrives at time a_j. The algorithm learns job j's all features upon the job's arrival. Competitive analysis is used to measure the performance of online algorithms in the worst case scenarios. If the objective is to be maximized, we say that an *online* algorithm is c-competitive if the algorithm's objective is at least c times the *offline* optimal scheduler's objective for *all* inputs. Equivalently, the algorithm is said to have a c-competitive ratio. Thus, competitive analysis does not make assumptions on the input and gives a performance guaranteed even in the worst case. The worst case analysis is of fundamental interest as the performance guarantee is independent of the input. However, when it is impossible to have a good upper bound on the competitive ratio, one could add certain parameters that constrain the inputs and seek to derive a competitive ratio depending on the parameters.

A. Applications

The multidimensional coflow (MD-Coflow) problem captures and unifies the following two well-studied problems.

1) Coflow: Consider a network fabric with $m \leq n$ ingress/egress ports. Each port has a certain capacity. There is a set of coflows to be served. Coflow j has arrival time a_j and may have to be satisfied by deadline b_j. Coflow j has demand vector $\{d_{uvj}\}_{u \in [m], v \in [n]}$ where d_{uvj} is the amount of data that coflow j must transfer from ingress port u to egress port v. A feasible schedule at each time is a collection of flows from ingress ports to egress ports subject to the capacity constraint of each port. Formally, if coflow j sends f_{uvjt} units of packets from ingress port u to egress port v at time t, it must satisfy $\sum_{v,j} f_{uvjt} \leq c(u)\ \forall u, t$ and $\sum_{u,j} f_{uvjt} \leq c(v)\ \forall v, t$ where $c(u)$ and $c(v)$ denote the capacity of u and v, respectively.

Coflow is an elegant abstraction to model the communication stage of various types of data-parallel applications in modern massively parallel computing platforms such as MapReduce [2] and Spark [3]. As a result of its faithfulness to real-world scheduling and clean abstraction, coflow has received significant attentions [1], [16]–[20] since its introduction [1]. Coflow has been studied for various objectives such as minimizing total completion time and throughput.

To see MD-Coflow indeed generalize the coflow, think of each port as a resource. There are two types of resources, ingress ports and egress ports. Map each coflow j to a job j, and for each pair (u, v) of ingress and egress ports, create a task q that demands resources u and v by an equal amount of d_{uvj}. Note that each task uses exactly two resources.

2) Multidimensional Scheduling: In this problem, there is a set $\mathcal{R}$ of resources and no two resources are of the same type. Each job j has exactly one task, and therefore, there is no need to distinguish between jobs and tasks. Job j has demand d_{jr} for each resource r. To process j, we need to assign all resources demanded by the job. If a job j is given h_{jr} units of every resource r *simultaneously* at a time, it is processed by a fraction of $\min_{r:d_{jr}\neq 0} h_{jr}/d_{jr}$[1]. In other words, each job j's processing rate is constrained by the resource the job is given least relative to its demand of the resource.

Multidimensional scheduling has recently received significant attentions as it allows a more effective scheduling when jobs require heterogeneous resources. Indeed, in the aforementioned massively parallel platforms, a task could be IO-intensive or CPU-intensive, or could be a certain mixture of both. Multidimensional scheduling has been studied in both offline and online settings for various objectives such as total completion time and fairness to the clients [9]–[13], [21], [22]. Particularly, Zheng and Shroff [8] recently consider partial throughput for this problem.

At a high-level, coflow intends to capture parallel execution/communication while the multidimensional scheduling allows a better utilization of heterogeneous resources that cannot be substituted. Considering both views simultaneously has some advantages. For example, it could be beneficial to consider computation and communication simultaneously when some jobs, while being processed, produce partial outcomes that can be useful for the next round computation. Thus, the combination of the two popular models provides a rich semantics.

III. Other Related Work

Coflow and multidimensional scheduling have been extensively studied, and hence, our discussion of previous work is inherently incomplete. Here, we only discuss some of the most related work that is not covered in the previous section. Partial throughput has been considered for homogeneous resources prior to the work [8] by Zheng and Shroff on heterogeneous resources. For the homogeneous (single) resource case, concave utility functions [4], [7] , as well as linear utility functions, were considered [23]–[25]. The standard throughput maximization where only fully executed jobs are counted has been extensively studied, e.g. [26]–[28]. However, only special cases admit constant competitive algorithms. To get around the barrier, in [28], it was assumed that job sizes are considerably smaller than their lifespan length.

IV. Our Algorithm and Theoretical Results

A. Myopic Concurrent Algorithms

In this paper, we focus on a natural class of schedules, which we call *concurrent*, that process tasks of the same job at an equal rate. Formally,

Definition 1 (Concurrent Schedules)**.** We say that a schedule is concurrent if, at any point in time, all tasks of each job are processed at an equal rate, i.e., for all $j \in J$, $t \in [a_j, b_j]$, $q \neq q' \in Q_j$, $y_{qt} = y_{q't}$.

[1]In general, a job j can have a specific utility function that takes an input resource vector and outputs the job's processing rate. This utility function is called Leontief and is one of the most popular utilities considered in the literature.

We will study a special class of algorithms that produce concurrent schedules in a myopic way. Thus, we will call such algorithms *myopic concurrent*. Specifically, a myopic concurrent algorithm seeks to *maximize the marginal partial throughput* at each time t *ignoring jobs deadlines*. Let J_t denote the set of jobs that are alive at time t, i.e., $a_j \leq t \leq b_j$. At each time slot t, given the progress of each job in previous time slots $\{z_{jt'}\}_{t'<t}$, the algorithm maximizes the marginal improvement by solving the following convex program.

$$\begin{aligned}
\max \quad & \sum_{j\in J_t} f_j\Big(\sum_{\tau=1}^{t} z_{j\tau}\Big) && \mathsf{CP}_{\text{CON-MYOPIC}} \qquad (5)\\
\text{s.t.} \quad & \sum_{j\in J_t, q\in Q_j} x_{qrt} \leq 1 && \forall r \in \mathcal{R}\\
& x_{qrt} \geq d_{qr} z_{jt} && \forall r \in \mathcal{R}, j \in J_t, q \in Q_j\\
& x_{qrt} \geq 0 && \forall r \in \mathcal{R}, j \in J_t, q \in Q_j\\
& z_{jt} \geq 0 && \forall j \in J_t, q \in Q_j
\end{aligned}$$

Note that, $\mathsf{CP}_{\text{CON-MYOPIC}}$ is a continuous convex optimization problem, and therefore, can be solved efficiently. At each time t, the algorithm finds $\{z_{jt}\}_j$s by solving $\mathsf{CP}_{\text{CON-MYOPIC}}$. Then, for each task $q \in Q_j$, the algorithm assigns $z_{jt}d_{qr}$ units of each resource r to task q, meaning that at any point in time, the algorithm serves all tasks of the same job by an equal fraction of z_{jt}. Thus, the algorithm yields a concurrent schedule. It is easy to see that setting $y_{qt} = z_{jt}$ for all $t, j \in J_t, q \in Q_t$, we get a feasible solution to $\mathsf{CP}_{\text{MD-Coflow}}$.

Clearly, our myopic concurrent algorithm is online as its scheduling decision at each time does not depend on jobs arriving in the future. We will often call our algorithm $\mathcal{A}$.

B. Theoretical Results

We first analyze our algorithm in terms of a certain key parameter, which we introduce and call the *price of concurrence* (PoC). In the definition of PoC, we use the notion of speed augmentation [29]. We say that a schedule is s-speed if every resource is used by at most s units at any point time — more precisely, Constraint (1) becomes $\sum_{j,q\in Q_j} x_{qrt} \leq s$, increasing the capacity to s from 1. Intuitively, the PoC measures the minimum speed required for a concurrent schedule to process every job j as much as an optimal schedule, which is not necessarily concurrent, does.

Definition 2 (Price of Concurrence)**.** We say that a given multidimensional coflow instance has a price of concurrence (PoC), ρ if, for any feasible schedule, $\{x^*, y^*, z^*\}$ (feasible solution to $\mathsf{CP}_{\text{MD-Coflow}}$), there is a concurrent ρ-speed schedule $\{x, y, z\}$ such that $z_j \geq z_j^*$ for all jobs j.

Our main theorem shows our myopic concurrent algorithm $\mathcal{A}$'s performance is pretty close to the best performance one can hope for from any concurrent schedules. In other words, by restricting our schedules to concurrent schedules, we are at most ρ factor off from the optimal offline schedule and lose at most one additional factor by making scheduling decisions online. Thus, our online algorithm $\mathcal{A}$'s partial throughput is at least $1/(1+\rho)$ times the partial throughput of the optimal offline schedule. for all instances of PoC ρ.

Theorem 1 (Competitive Analysis of the Myopic Concurrent Algorithm)**.** *For all instances of PoC ρ, the myopic concurrent*

algorithm $\mathcal{A}$ is $1/(\rho+1)$-competitive for maximizing partial throughput.

We note that our result extends the previous work [8] that gives a 1/2-competitive algorithm that focuses on the multidimensional setting for the partial throughput objective — in other words, each job has only one task[2]. Thus, their setting implies $\rho = 1$, and therefore, our result immediately gives their result. It is known that no online algorithm can have a better than 1/1.25-competitive even for the single resource case [24].

We now turn our attentions to understanding the PoC. In the following, we give an upper bound on the parameter. In the definition of θ, the numerator is the maximum usage of any resource needed to complete job j, and the denominator is the average usage of resources of the same type needed to do so. Thus, we term θ the *maximum-to-average resource demand*, for short, the *MA parameter*.

Theorem 2 (Upper Bound on the Price of Concurrence)**.** *The price of concurrence ρ is at most the MA parameter, $\theta := \max_{j,k,r} \frac{\sum_{q\in Q_j} d_{qr}}{D_{jk}/|\mathcal{R}_k|}$.*

We note that $\theta = \max_{j,u,v} \frac{mn\cdot d_{uvj}}{\sum_{u,v} d_{uvj}}$ in coflow[3] and $\theta = \max_{j,r',r:d_{jr}\neq 0} \frac{d_{jr'}}{d_{jr}}$ in multidimensional scheduling[4].

V. Analysis

A. Competitive Analysis

This section is devoted to proving Theorem 1. We analyze the performance of our myopic concurrent algorithm $\mathcal{A}$ by interpreting the underlying scheduling problem as a submodular maximization subject to a partition matroid. Hence, we first take a detour to give a brief overview of submodular functions and matroids. Then, we discuss our interpretation in detail and complete our analysis.

1) Submodular Functions and Matroids: In this section, we give a quick overview of submodular functions and matroids. The reader who is interested in the rich theory of submodularity and matroids are referred to [30].

Definition 3. A set function $g : 2^U \to \mathbb{R}$ is submodular if $g(A\cup B) + g(A\cap B) \leq g(A) + g(B)$ for all $A, B \subseteq U$.

It is an easy exercise to show the following are alternative definitions of submodular functions.

Proposition 3. *A set function $g : 2^U \to \mathbb{R}$ is submodular if and only if $g(A\cup B) - g(A) \leq g(A'\cup B) - g(A')$ for all $B \subseteq U$ and $A' \subseteq A \subseteq U$.*

Proposition 4. *A set function $g : 2^U \to \mathbb{R}$ is submodular if and only if $g(B\cup\{e\}) - g(B) \leq g(A\cup\{e\}) - g(A)$ for $A \subseteq B \subseteq U$ and $e \in U$.*

Intuitively, submodularity captures diminishing marginal gains — the increase of one's happiness when acquiring a new item could be only smaller if she had more items. Define $g_A(B) := g(A\cup B) - g(A)$. For notational convenience, for an element in U, let $g_A(e) := g_A(\{e\})$. Likewise, we may drop $\{\}$ from $\{e\}$ to simplify the notation.

Definition 4. A set function $g : 2^U \to \mathbb{R}$ is monotone if $g(A) \leq g(B)$ for all $A \subseteq B \subseteq U$.

In this paper, we will be concerned with monotone submodular functions with non-negative values.

A matroid $\mathcal{M} = (U, \mathcal{I})$ is defined by a collection $\mathcal{I}$ of independent sets over a universe U of elements that satisfies the following properties: $\emptyset \in \mathcal{I}$; if $A \subseteq B$ and $B \in \mathcal{I}$, then $A \subseteq \mathcal{I}$; and if $A, B \in \mathcal{I}$ and $|A| < |B|$, then $\exists e \in B \setminus A$ such that $\{e\} \cup A \in \mathcal{I}$.

Matroids have many interesting properties and are found in many combinatorial problems. In this paper, we will particularly be interested in partition matroids, which are defined as follows.

Definition 5. Let $\{U_i\}_i$ be a partition[5] of U and $\mu_i \geq 0$ be a capacity for U_i. If a set of elements, S, is independent, i.e., $S \in \mathcal{I}$ if and only if $|S \cap U_i| \leq \mu_i$, then the collection $\mathcal{I}$ of independent sets forms a partition matroid.

The following theorem was shown in [31], but for completeness, here, we give a self-contained simple proof.

Theorem 5. *[31] Let g be a monotone submodular function, and let $\mathcal{M}$ be a partition matroid. Then, the following greedy algorithm is a 1/2-approximation for maximizing g subject to $\mathcal{M}$: Start with $S = \emptyset$ and repeatedly add to the current set S an element $e \notin S$ from an arbitrary U_i that maximizes $g(S\cup e) - g(S)$, ensuring that S stays independent under $\mathcal{M} = (U, \mathcal{I})$, i.e., $S \in \mathcal{I}$.*

Proof: We show the theorem assuming that $\mu_i = 1$ for all i. This is because for our purposes, showing this special case will be sufficient and extending the analysis to arbitrary μ_i is straightforward. By reindexing, assume w.l.o.g. that the greedy algorithm adds elements in this order $e_1, e_2, ..., e_T$. Since g is monotone and $\mu_i = 1$ for all i, we can assume w.l.o.g. that we choose exactly one element from each U_i; we assume w.l.o.g. that $U_i \neq \emptyset$. Let A and O denote the greedy algorithm and the optimal solution, respectively. Let e_i and e_i^* denote the elements A and O choose from U_i. Let $E_t := \{e_1, e_2, ..., e_t\}$ and $E_t^* = \{e_1^*, e_2^*, ..., e_t^*\}$. Then, we have,

$$
\begin{aligned}
g(E_T) - g(\emptyset) &= \sum_{t=1}^{T} g(E_t) - g(E_{t-1}) \\
&\geq \sum_{t=1}^{T} g(E_{t-1}\cup e_t^*) - g(E_{t-1}) \qquad (6) \\
&\geq \sum_{t=1}^{T} g(E_T \cup E_{t-1}^* \cup e_t^*) - g(E_T \cup E_{t-1}^*) \qquad (7) \\
&= \sum_{t=1}^{T} g(E_T \cup E_t^*) - g(E_T \cup E_{t-1}^*) = g(E_T \cup E_T^*) - g(E_T)
\end{aligned}
$$

[2]Their setting is slightly different from ours as in their work a job consists of multiple identical tasks, but the tasks of the same job in their work can be viewed as one task in our work

[3]This parameter is derived after we make $c(u) = c(v) = 1$ for all ports u, v by scaling.

[4]One can alternatively assume that all resources are of the same type. Then, $\theta = \max_{j,r',r}(|\mathcal{R}|d_{jr'})/(\sum_r d_{jr})$

[5]That is, for any $i \neq i'$, $U_i \cap U_{i'} = \emptyset$ and $\cup_i U_i = U$.

Eq. (6) holds true as the greedy algorithm chose element $e_t \in U_t$ since $g_{E_{t-1}}(e_t) \geq g_{E_{t-1}}(e)$ for all $e \in U_t$. Eq. (7) follows from the fact that g is submodular and Proposition 3. By rearranging terms, we have $g(E_T) \geq \frac{1}{2} g(E_T \cup E_T^*) \geq \frac{1}{2} g(E_T^*)$ where the last inequality follows from the monotonicity of g. ∎

2) Interpreting (Partial) Throughput Maximization as Submodular Maximization: We now show that a large class of (partial) throughput maximization problems can be interpreted as a submodular maximization subject to a partition matroid, which admits a simple 1/2-approximation as we saw in Theorem 5. Let $z_{j,t}$ denote how much job j gets processed at time t; so $z_{jt} = 0$ for all $t < a_j$. Each job j is associated with a utility function $f_j : [0, \infty) \to [0, \infty)$. Let $z_{j,\leq t} = z_{j,a_j} + z_{j,a_j+1} + \cdots + z_{j,t}$; we keep commas between subscripts when they improve the readability. Consider the following optimization problem.

$$\max \sum_j f_j(z_{j,\leq T}) \text{ s.t. } \{z_{j,t}\}_j \in \text{ a convex body } \mathcal{P}_t \forall t, \quad (8)$$

where a feasible schedule at each time t is within a convex body $\mathcal{P}_t$ over $\{z_{jt}\}_j$. Note that job sizes can be implicitly captured by f_j by setting $f_j(p) = 0$ for all $p \geq p_j$ where p_j denotes j's size.

Now let's interpret this problem as a submodular maximization problem subject to a partition matroid. Towards this end, we first create the universe of the elements in the matroid. Let S_t denote the set of all[6] possible schedules at time t. For a schedule $s \in S_t$, let $z_{jt}(s)$ denote how much the job gets processed at time t under the scheduler s. Define $U := \{(t, s) \mid t \in [0, T] \cap \mathbb{Z} \text{ and } s \in S_t\}$. Define $U_t := \{(t, s) \mid s \in S_t\}$. Note that $\{U_t\}_t$ is a partition of U. We say that a subset $S \subseteq U$ is independent if and only if $|S \cap U_t| \leq 1$ for all t. Note that this forms a partition matroid, which we denote as $\mathcal{M}$. Then, a feasible schedule over time period $[0, T]$ corresponds to a maximal independent set of matroid $\mathcal{M}$.

Now we create a submodular function $F : 2^U \to [0, \infty)$ to capture the above optimization problem. Intuitively, a subset of elements from U defines a schedule that dictates the objective; here the schedule is not necessarily feasible. Formally, $F(S)$ is defined as follows. Let $z_{jt}^S = \sum_{(s,t) \in U_t \cap S} z_{jt}(s)$ and $F(S) = \sum_j f_j(z_{j,\leq T}^S)$ where $z_{j,\leq t}^S = z_{j,a_j}^S + ... + z_{j,t}^S$.

Lemma 6. *The function F is submodular.*

Proof: We prove F's submodularity by showing that F satisfies the condition stated in Proposition 4. Consider arbitrary two sets $A \subseteq B \subseteq U$ and an element $e = (t_0, s) \in U$. Assume that $e \notin B$ since the claim otherwise holds immediately. Let $\Delta_j := z_{j,t_0}(s)$; Δ_j is the amount of j processed by the schedule/element e at time t_0. Then, we have,

$$F(B \cup \{e\}) - F(B) = \sum_j \Big(f_j(z_{j,\leq T}^B + \Delta_j) - f_j(z_{j,\leq T}^B) \Big)$$

$$\leq \sum_j \Big(f_j(z_{j,\leq T}^A + \Delta_j) - f_j(z_{j,\leq T}^A) \Big) = F(A \cup \{e\}) - F(B),$$

where the inequality follows since $A \subseteq B$, and thus, $z_{j,\leq T}^B \geq z_{j,\leq T}^A$, and f_j is concave. ∎

Then the greedy algorithm stated in Theorem 5 immediately translates into an algorithm that optimizes the following at each time t. Note that $\sum_j f_j(z_{j,\leq t-1})$ is fixed, and therefore, we can drop the quantity from the objective.

$$\max \sum_j f_j(z_{j,\leq t}) - \sum_j f_j(z_{j,\leq t-1}) \quad \text{s.t.} \{z_{j,t}\}_j \in \mathcal{P}_t \ \forall t \quad (9)$$

Further, this is an online algorithm since the greedy algorithm can consider times in increasing order and only needs to know which jobs are alive at each time and the convex body to which $\{z_{j,t}\}_j$ are subject. Thus, this is an online algorithm and is 1/2-competitive by Theorem 5.

3) Interpreting Concurrent Partial Throughput Maximization as Submodular Maximization: The problem of maximizing partial throughput for the multidimensional coflow doesn't fit into the framework that is described in the convex program (8). However, we observe that the multidimensional coflow actually does when the schedule is required to be concurrent. Then, the concurrent multidimensional coflow can be expressed as the following convex program.

$$\begin{aligned}
&\max \sum_j f_j(\sum_t z_{j,t}) \\
\sum_{j,q \in Q_j} x_{qrt} &\leq 1 && \forall r \in \mathcal{R}, t \\
x_{qrt} &\geq d_{qr} z_{jt} && \forall r \in \mathcal{R}, j \in J, q \in Q_j, t \\
x_{qrt} &\geq 0 && \forall r \in \mathcal{R}, j \in J, q \in Q_j, t \\
z_{jt} &= 0 && \forall j \in J, t \notin [a_j, b_j] \\
z_{jt} &\geq 0 && \forall j \in J, t
\end{aligned}$$

Since the schedule is required to be concurrent, to process job j by a fraction of δ, task $q \in Q_j$ needs to receive $d_{qr}\delta$ units of resource r. Job j is processed by a fraction of $\sum_t z_{jt}$ throughout the schedule. Note that we can remove the constraint $z_j \leq 1$ by capping f_j at $f_j(1)$, i.e., $f_j(z) = f_j(1)$ when $z \geq 1$; function f_j remains monotone and concave and by smoothing we can assume that f_j is differentiable. Then, we note that the set of feasible schedules at time t are subject to a convex body that is defined by the above constraints. Observe that the myopic concurrent algorithm $\mathcal{A}$ is exactly solving the convex program (9) at each time t.

Corollary 7. The concurrent myopic greedy algorithm yields a schedule that gives at least half the maximum partial throughput any concurrent schedule does.

So, the corollary, by the definition of PoC, implies that the concurrent myopic greedy algorithm $\mathcal{A}$ is $1/(2\rho)$-competitive. In fact, we can further refine the performance of our algorithm as follows. We similarly define U^* for the concurrent optimal

[6]As a technical note, we note that under some reasonable assumptions, a convex body can be approximated by a polytope to an arbitrary precision, e.g., [32]. Thus, the set of vertices of the polytope provides all scheduling choices to an arbitrary precision, which allows us to assume that S_t is finite. This is only for the sake of analysis and our algorithm does not require S_t to be finite.

schedule as we defined U for our schedule. Formally, S_t^* denote the possible set of scheduling decisions the concurrent optimal scheduler can make at time t. There is one-to-one correspondence schedules in S and those in S^* and a schedule in S^* processes jobs ρ times as much as its corresponding schedule in S does. All sets $\{U_t^*\}$ are defined analogously. We extend the domain of F to $U \cup U^*$. The proof of Lemma 6 is literally the same, except that elements are now from $U \cup U^*$; thus, F is submodular over the extended domain $U \cup U^*$.

We now revisit the proof of Theorem 5. As before, let e_t denote the element corresponding to the schedule our algorithm makes at time t, and $E_t := \{e_1, e_2, \cdots, e_t\}$; note that $A \subseteq U$. Likewise, e_t^* is the element in $U_t^* \subseteq U^*$ corresponding to the schedule the concurrent optimal scheduler makes at time t, and $E_t^* := \{e_1, e_2, \cdots, e_t\}$. In the proof of Theorem 5, we only need to slighly modify Eqn. (7): we now have $F(E_t) - F(E_{t-1}) \geq \frac{1}{\rho}\left(\sum_{t=1}^{T} F(E_{t-1} \cup e_t^*) - F(E_{t-1})\right)$. This holds since our algorithm $\mathcal{A}$ could have chosen a schedule e_t' in U_t corresponding to $e_t^* \in U_t^*$. Thus, $F(E_t) - F(E_{t-1}) \geq F(E_{t-1} \cup e_t') - F(E_{t-1}) = \frac{1}{\rho}(F(E_{t-1} \cup e_t^*) - F(E_{t-1}))$, where the last equality holds because a time slot is infinitesimally small, all functions f_j are differentiable, and e^* processes every job exactly ρ times as much as e_t' does. Then, the rest of the proof remains the same, and after rearranging terms, we derive $F(E_T) \geq \frac{1}{\rho+1} F(E_T^*)$. This completes the proof of Theorem 1.

B. Analysis of Price of Concurrence

This section is devoted to proving Theorem 2. To this end, we will prove that if there is a schedule σ_1 where each job j is processed by z_j fraction, then there is a θ-speed concurrent schedule σ_2 where each job j is processed by as much. Assume w.l.o.g. that in the given schedule σ_1, each job j consumes resource r by exactly $z_j \sum_{q \in Q_j} d_{qr}$ units during $[a_j, b_j]$ since a task $q \in Q_j$ using resource r by more than $z_j d_{qr}$ units would waste resource r. Thus, job j consumes $z_j D_{jk}$ amount of resources of type k during its lifespan, $[a_j, b_j]$.

Fix an arbitrary type k^*. We can abstract and represent σ_1 as follows. Let X_{jt} denote the total amount of resources of type k^* that job j uses at time t. Then, we have,

$$\sum_{t \in [a_j, b_j]} X_{jt} = z_j D_{jk^*} \qquad \forall j \tag{10}$$

$$\sum_{j} X_{jt} \leq |\mathcal{R}_{k^*}| \qquad \forall t \tag{11}$$

where $X_{jt} \geq 0$. Eq. (10) is a restatement of the above discussion. Each job uses resources of type k^* by X_{jt} units, thus total usage of resources of the type is upper bounded by $|\mathcal{R}_{k^*}|$, which is the total units of resources of type k^* at each time. Thus, we have Eq. (11).

By letting $X'_{jt} = X_{jt}/|\mathcal{R}_{k^*}|$, we have,

$$\sum_{t \in [a_j, b_j]} X'_{jt} = \frac{z_j D_{jk^*}}{|\mathcal{R}_{k^*}|} \qquad \forall j \tag{12}$$

$$\sum_{j} X'_{jt} \leq 1 \qquad \forall t \tag{13}$$

where $X'_{jt} \geq 0$. Since a time slot is infinitesimally small, one can assume that $X'_{jt} \in \{0, 1\}$ by setting $X'_{jt'} = 1$ for $X'_{jt} dt$ time units during an infinitesimally small interval $[t, t + dt)$. We can view $\{X'_{jt}\}_{j,t}$ as a schedule in the standard single server setting where job j is processed at time t if an only if $X'_{jt} = 1$. Note that at each time exactly one job or no job is processed. Thus, as a result of Eq. (12), each job j is processed by $z_j D_{jk^*}/|\mathcal{R}_{k^*}|$ units during its lifespan $[a_j, b_j]$.

We are now ready to construct the desired schedule, σ_2. Fix an infinitesimally small interval $[t, t + dt)$. When job j is processed during $[t, t + dt)$ in σ', i.e., $X'_{jt} = 1$, we give to each task $q \in Q_j$, $\frac{d_{qr}}{D_{jk^*}/|\mathcal{R}_{k^*}|} dt$ units of each resource r in schedule σ_2 Then, resource r is used by $\frac{\sum_{q \in Q_j} d_{qr}}{D_{jk^*}/|\mathcal{R}_{k^*}|} dt$ units during $[t, t + dt)$ as job j is the only job that is processed at time t since $X'_{j't} = 0$ for all $j \neq j'$ due to Eq. (13). This quantity is upper bounded by $\theta \cdot dt$ by definition of θ. Thus, σ_2 is a feasible schedule with a θ-speed. Since all tasks $q \in Q_j$ receives resources r in proportion to their demand d_{qr}, σ_2 is concurrent. Further, task q receives exactly $\frac{d_{qr}}{D_{jk^*}/|\mathcal{R}_{k^*}|} \cdot \frac{z_j D_{jk^*}}{|\mathcal{R}_{k^*}|} = d_{qr} z_j$ units of resource r during its lifespan. Thus, job j is processed exactly by z_j fraction in σ_2. Hence, σ_2 is a feasible schedule that completes each job j by z_j fraction when given a θ-speed. This implies the PoC of the given instance is upper bounded by θ, as desired.

VI. Experiments

In this section, we evaluate the performance of our online coflow scheduling algorithm with numerical studies. We show that our algorithm achieves clearly higher utility gains compared with several heuristics adapted from existing coflow scheduling algorithms developed for different objectives.

Workload: We consider a similar simulation setting as in [18], [33], [34]. A 50×50 network switch is considered. Each ingress/egress port has a capacity of 1 Gbps. We set a time slot to be 1/128 second so that each port can transmit 1 MB data per time slot. 100 coflows are generated. Coflows arrive at the beginning of a second. The number of new arrivals in each second follows a Poisson distribution with mean λ, independent of other seconds. For each coflow, the number of mapper tasks and the number of reducer tasks, each follows a uniform distribution in [minWidth,maxWidth], where no two mappers (reducers) share an ingress (egress) port. For every pair of mapper and reducer specified in a coflow, there is a flow in between. The traffic of each flow in a coflow is generated as follows. As in [18], we assume that the shuffle data of each reducer in a coflow is evenly generated from all mappers of that coflow. For each reducer in a coflow, the amount of shuffle data from each mapper of the coflow is sampled from a uniform distribution in [minLength,maxLength] (MB). Note that the parameters minWidth, maxWidth, minLength, maxLength together determine the distribution of traffic across ports. Moreover, deadlines for coflows are generated in a similar way as in [18]. For each coflow, we first determine its minimum coflow completion time (CCT) in an empty switch. The allowed duration of the coflow is then defined as its CCT multiplied by $1 + \mathcal{U}(0, x))$, where $\mathcal{U}(0, x)$ denotes a uniform distribution between 0 and a parameter x. The utility gain of a coflow that has been processed by z_j fraction by its deadline is modeled by $f_j(z_j) = \log(1 + z_j)$. We also considered other concave functions and observed similar

results. We generate 100 instances for each set of parameters and report the algorithms' average performance.

Algorithms: We compare our algorithm with several heuristics in simulations. One challenge to implement our algorithm in a real setting is that solving a convex optimization problem in each small time slot can be time consuming. We consider the following approximation to the algorithm to reduce the overhead. A new scheduling decision is made only when one of the following events happens, (a) the current time slot is the first time slot in a second, (b) there are new arrivals in this time slot, and (c) there are departures in the previous time slot. When making a decision, a convex optimization problem is formulated by pooling together all the time slots before the next departure of any active coflow or the next second. The solution to the convex program gives the rate for serving each active coflow, which is updated until the next event happens.

In addition to our algorithm, we consider the following algorithms in simulations. They are adapted from the coflow scheduling algorithms proposed in [18] by taking deadlines and partial values into consideration.

ADMIT (admission control): This is the algorithm proposed in [18] for deadline constrained coflows without partial values. In this algorithm, a newly arrived coflow is admitted only if it can meet its deadline without hurting existing coflows in the switch. Each flow in a coflow is given the minimum capacity needed to finish the flow by its deadline. The remaining bandwidth is distributed to active coflows using a backfilling procedure. Thus, once admitted, a coflow is never preempted and is guaranteed to finish by its deadline. Hence each coflow is either fully served or rejected.

FAIR: In this algorithm, flows from all active coflows are served with an equal rate. This algorithm provides fairness at the flow level (but not the coflow level).

In the following two algorithms, all the coflows are admitted to the system. In both algorithms, scheduling decisions are made when there are arrivals or departures, and the decisions are made for each active coflow one by one according to some priority ordering of coflows. For each coflow, all the flows in the coflow are served in the same rate so that the coflow can finish as soon as possible using the remaining bandwidth available.

FIFO (First-In-First-Out): This algorithm sorts active coflows by their arrival times.

SEBF (Smallest-Effective-Bottleneck-First): In this algorithm, coflows with smaller effective bottleneck are given higher priority, where the effective bottleneck of a coflow is defined as its minimum CCT in an empty network. This is the algorithm used by Varys [18] for minimizing CCT.

To adapt these algorithms to our problem, an unfinished coflow is removed from the system on its deadline with its partial value retained.

Results: In Figure 1(a), we plot the algorithms' utility gains under different deadline constraints. We vary the value of x from 0 to 4 and fix other parameters. We observe that all the algorithms obtain higher utility gains when the workload becomes more time elastic. Our algorithm achieves about 30% higher utility compared to all the heuristics. SEBF, FIFO, and FAIR have similar performance while ADMIT has the worst performance due to the ignorance of partial values.

In Figure 1(b), we vary the mean arrival rate λ and fix other parameters. The utility gains of all the algorithms decrease with large arrival rates as expected. Compared with SEBF, FIFO and FAIR, our algorithm obtains more advantage for large arrival rates. Intuitively, our algorithm tends to distribute the capacity more evenly among coflows, which gives higher total utility gain when there are more active coflows competing for the capacity. We also observe that the gap between ADMIT and other policies become smaller for large arrive rates.

In both cases discussed above, we have fixed [minWidth, maxWidth] = $[10, 20]$. In Figure 1(c), we compare the algorithms' performance with [minWidth, maxWidth] = $[10, 20]$ and [minWidth, maxWidth] = $[1, 20]$, respectively. In the latter case, each coflow has less number of mappers and reducers on overage, that is, the workload is sparser. Also, the distribution of coflow size is more diverse in the latter case. Note that with other parameters fixed, the sparser the workload is, the larger the maximum-to-average demand θ is. Thus, we expect the greedy algorithm to have a larger optimality gap for the sparser workload. This is confirmed in Figure 1(c). In particular, the improvement of our algorithm over the three heuristics is smaller for the sparser workload. Furthermore, SEBF outperforms FIFO and FAIR in the second scenario. This is because when the coflow size distribution becomes more diverse, large coflows can block multiple smaller ones under the FIFO policy, which hurts the total utility (since all the coflows have the same utility function in our simulation), while SEBF gives higher priority to smaller coflows. Under the FAIR policy, large coflows are given more bandwidth than small ones, which hurts the total utility gain when the coflow size distribution becomes more diverse.

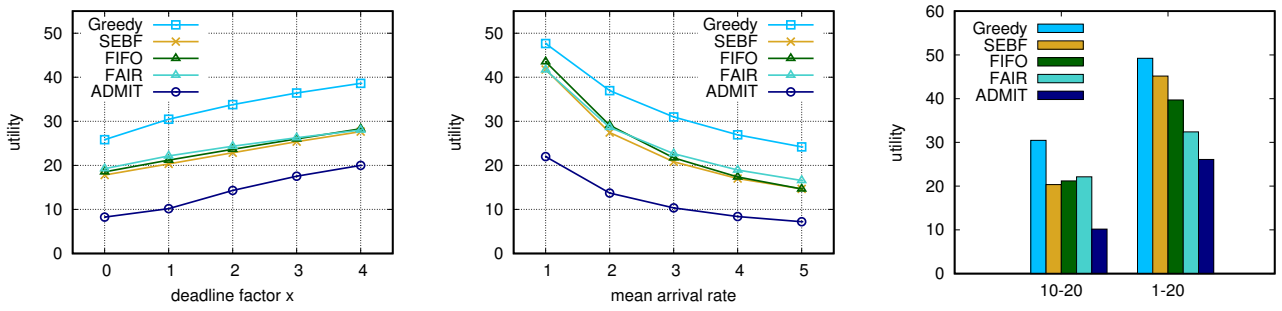
VII. Conclusions

In this paper, we considered a general model that naturally combines coflow and multidimensional scheduling and studied the problem for maximizing partial throughput to measure the values of partially completed jobs/coflows. We showed an online greedy algorithm that schedules jobs (or tasks) at each time such that the marginal increase of the objective is maximized has a theoretical performance guarantee that is close to that of the best concurrent optimal schedule that processes all tasks of the same job at an equal rate. Also, we showed the gap (price of concurrence) between concurrent schedules and the optimal schedule is upper bounded by the MA parameter that is easy-to-compute.

We close with some interesting future research directions. In this paper, we gave an upper bound on the price of concurrence but we are not aware of any interesting lower bounds on the PoC. Also, it would be interesting to study multidimensional coflow for various objectives other than partial throughput.

Acknowledgment

Im and Shadloo are supported in part by NSF grants CCF-1409130 and CCF-1617653.



(a) (b) (c)

Fig. 1. Simulation Results. In (a), $\lambda = 3$, the job allowance of a coflow j, that is $b_j - a_j$, equals to its minimum CCT multiplied by $1+\mathcal{U}(0, x)$. In (b), $x = 1$. In both (a) and (b), [minWidth, maxWidth] = [10,20], [minLength, maxLength] = [10,20]. In (c), $\lambda = 3$, $x = 1$, [minLength,maxLength] = [10,20]. In the left part of (c), [minWidth, maxWidth] = [10,20]. In the right part of (c), [minWidth, maxWidth] = [1,20].

REFERENCES

[1] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *ACM HotNets*. ACM, 2012, pp. 31–36.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[4] Y. He, S. Elnikety, J. Larus, and C. Yan, "Zeta: Scheduling interactive services with partial execution," in *SoCC*. ACM, 2012, p. 12.

[5] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics," in *Proc. of NSDI*, 2014.

[6] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *Proc. of ASPLOS*, 2015.

[7] Y. Zheng, B. Ji, N. Shroff, and P. Sinha, "Forget the deadline: Scheduling interactive applications in data centers," in *CLOUD*. IEEE, 2015, pp. 293–300.

[8] Z. Zheng and N. B. Shroff, "Online multi-resource allocation for deadline sensitive jobs with partial values in the cloud," in *INFOCOM*. IEEE, 2016, pp. 1–9.

[9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, vol. 11, no. 2011, 2011, pp. 24–24.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[11] S. Im, J. Kulkarni, and K. Munagala, "Competitive algorithms from competitive equilibria: Non-clairvoyant scheduling under polyhedral constraints," in *ACM STOC*. ACM, 2014, pp. 313–322.

[12] S. T. Maguluri and R. Srikant, "Scheduling jobs with unknown duration in clouds," *IEEE/ACM Transactions on Networking (TON)*, vol. 22, no. 6, pp. 1938–1951, 2014.

[13] S. M. Zahedi and B. C. Lee, "Ref: Resource elasticity fairness with sharing incentives for multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 145–160, 2014.

[14] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. cambridge university press, 2005.

[15] C. Chekuri, A. Gal, S. Im, S. Khuller, J. Li, R. M. McCutchen, B. Moseley, and L. Raschid, "New models and algorithms for throughput maximization in broadcast scheduling," in *WAOA*, vol. 10. Springer, 2010, pp. 71–82.

[16] S. Ahmadi, S. Khuller, M. Purohit, and S. Yang, "On scheduling coflows," in *IPCO*. Springer, 2017, pp. 13–24.

[17] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *SIGCOMM*. ACM, 2015, pp. 393–406.

[18] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *SIGCOMM*. ACM, 2014, pp. 443–454.

[19] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards practical and near-optimal coflow scheduling for data center networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. PP, no. 99, pp. 1–1, 2016.

[20] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *INFOCOM*. IEEE, 2015, pp. 424–432.

[21] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multiresource allocation: Fairness-efficiency tradeoffs in a unifying framework," *IEEE/ACM TON*, vol. 21, no. 6, pp. 1785–1798, 2013.

[22] I. Kash, A. D. Procaccia, and N. Shah, "No agent left behind: Dynamic fair division of multiple resources," *Journal of Artificial Intelligence Research*, vol. 51, pp. 579–603, 2014.

[23] F. Y. Chin, M. Chrobak, S. P. Fung, W. Jawor, J. Sgall, and T. Tichỳ, "Online competitive algorithms for maximizing weighted throughput of unit jobs," *J. of Discrete Algorithms*, vol. 4, no. 2, pp. 255–276, 2006.

[24] F. Y. Chin and S. P. Fung, "Online scheduling with partial job values: Does timesharing or randomization help?" *Algorithmica*, vol. 37, no. 3, pp. 149–164, 2003.

[25] M. Chrobak, L. Epstein, J. Noga, J. Sgall, R. van Stee, T. Tichỳ, and N. Vakhania, "Preemptive scheduling in overloaded systems," *Automata, Languages and Programming*, pp. 777–777, 2002.

[26] N. Jain, I. Menache, J. S. Naor, and J. Yaniv, "Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters," *ACM Transactions on Parallel Computing*, vol. 2, no. 1, p. 3, 2015.

[27] G. Koren and D. Shasha, "D/sup over/; an optimal on-line scheduling algorithm for overloaded real-time systems," in *Real-Time Systems Symposium, 1992*. IEEE, 1992, pp. 290–299.

[28] B. Lucier, I. Menache, J. S. Naor, and J. Yaniv, "Efficient online scheduling for deadline-sensitive jobs," in *ACM SPAA*, 2013, pp. 305–314.

[29] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *Journal of the ACM (JACM)*, vol. 47, no. 4, pp. 617–643, 2000.

[30] A. Schrijver, *Combinatorial optimization: polyhedra and efficiency*. Springer-Verlag, Berlin, 2003.

[31] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions?i," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[32] E. M. Bronstein, "Approximation of convex sets by polytopes," *Journal of Mathematical Sciences*, vol. 153, no. 6, pp. 727–762, 2008.

[33] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *SPAA*. ACM, 2015, pp. 294–303.

[34] M. Shafiee and J. Ghaderi, "An improved bound for minimizing the total weighted completion time of coflows in datacenters," *arXiv preprint arXiv:1704.08357*, 2017.