Scheduling Coflows of Multi-stage Jobs to Minimize the Total Weighted Job Completion Time

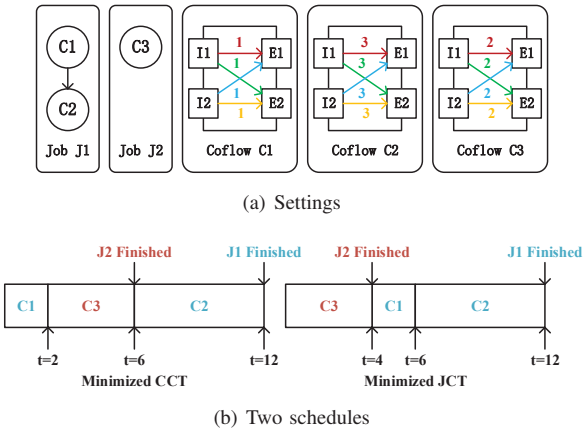
Bingchuan Tian, Chen Tian, Haipeng Dai, and Bingquan Wang
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China
{bctian,wangbingquan}@smail.nju.edu.cn, {tianchen,haipengdai}@nju.edu.cn

***Abstract*—Datacenter networks are critical to Cloud computing. The coflow abstraction is a major leap forward of application-aware network scheduling. In the context of multi-stage jobs, there are dependencies among coflows. As a result, there is a large divergence between coflow-completion-time (CCT) and job-completion-time (JCT). To our best knowledge, this is the first work that systematically studies: *how to schedule dependent coflows of multi-stage jobs, so that the total weighted job completion time can be minimized*. We present a formal mathematical formulation. We also prove that this problem is strongly NP-hard. Inspired by the optimal solution of the relaxed linear programming, we design an algorithm that runs in polynomial time to solve this problem with an approximation ratio of $(2M + 1)$, where M is the number of machines. Evaluation results demonstrate that, the largest gap between our algorithm and the lower bound is only 9.14%. We reduce the average JCT by up to 33.48% compared with Aalo, a heuristic multi-stage coflow scheduler. We reduce the total weighted JCT by up to 83.31% compared with LP-OV-LS, the state-of-the-art approximation algorithm of coflow scheduling.**

I. Introduction

Motivation: Datacenter networks are critical to Cloud computing. In modern datacenters, data-parallel frameworks (*e.g.*, MapReduce [1], Hadoop [2], Spark [3]) are widely used to run distributed computing jobs, such as querying and data mining. Data transfer has a significant impact on job performance. For example, a MapReduce/Hadoop job is scheduled by a master process to execute m mapper tasks and r reducer tasks. Each mapper task reads files from the underlying distributed file system (DFS), performs user-defined computations, and writes the outputs back to DFS. Each reducer task reads the output data of mappers, merges them and writes the final results to DFS. This data transmission phase is called as *shuffle*, where totally $m \times r$ flows are generated for the job. It is reported that sometimes, 50% of the job time is spent on transferring shuffle data across the networks [4].

The *coflow* abstraction is a major leap forward of application-aware network scheduling. Traditional network metrics, such as the average flow-completion-time (FCT), ignore application semantics of data-parallel jobs. For a shuffle in MapReduce/Hadoop, the completion time of the slowest flow (instead of the average FCT) dominates the start of reducer computation. Minimizing the average FCT does not necessarily minimize the job-completion-time (JCT). Being aware of this problem, a coflow is defined as the collection of all flows in a shuffle, and coflow-completion-time (CCT) is the completion time of the slowest flow in a certain coflow [5]. Current work focus on minimizing the average CCT. For jobs with a single shuffle phase (*e.g.*, Terasort), minimizing the average CCT usually results in faster jobs, because there are only one coflow and time spent on computation can be considered as nearly constant [6], [7], [8].



(a) Settings

(b) Two schedules

Fig. 1. A motivation example

In the context of multi-stage jobs, there are dependencies among coflows. In modern datacenters, it is common that a job contains more than one stages with dependencies. For example, each TPC-DS query (of distributed database applications) is a directed-acyclic-graph (DAG) of multi-stage dataflow [9]. As a result, a coflow C_2 can be dependent on another coflow C_1 in the same job if the consumer computation stage of C_1 is the producer of C_2. There are two kinds of dependencies: *Starts-After* and *Finishes-Before*. *Starts-After* represents the existence of explicit barriers (*e.g.*, the write barriers in Hadoop). In this case, C_2 cannot start until C_1 has finished. *Finishes-Before* is common for pipeline based frameworks (*e.g.*, Spark), where C_2 can coexist with C_1 but it cannot finish until C_1 has finished. In this paper we focus on scheduling coflows of *Starts-After* type multi-stage jobs, and leave *Finishes-Before* type jobs to future work.

There is a large divergence between CCT and JCT for multi-stage jobs. Consider a motivation example in Fig. 1. There are two equal-weight jobs J_1 and J_2 arrived and waiting to be scheduled. J_1 has 2 coflows C_1 and C_2 with a dependency that C_2 cannot start until C_1 has finished (*i.e.*, *Starts-After*), while J_2 has only one coflow C_3. $C_1/C_2/C_3$ each has 4 flows. The flows are of 1/3/2 unit(s) size for $C_1/C_2/C_3$ respectively as shown in Fig. 1(a). The total sizes of these coflows are 4/12/8 units, respectively. The topology is non-blocking, and each

978-1-5386-4128-6/18/$31.00 ©2018 IEEE

input/output port can accept one unit flow size in one time step. It takes 2/6/4 steps for $C_1/C_2/C_3$ to pass the network bottleneck if occupied exclusively. The minimal average CCT is $\frac{2+(2+4)+(2+4+6)}{3} \approx 6.67$ if we schedule coflows in the order of (C_1, C_3, C_2). The corresponding average JCT is $\frac{(2+4+6)+(2+4)}{2} = 9$. However, if we schedule the coflows in the order of (C_3, C_1, C_2), the average CCT increases to $\frac{4+(4+2)+(4+2+6)}{3} \approx 7.33$, while the average JCT decreases to $\frac{4+(4+2+6)}{2} = 8$. Note that in this example we assign equal weights to J_1 and J_2. Usually in production systems, important jobs are prioritized by setting a larger weight value. Accordingly, we extend the optimization objective from the average JCT to the total weighted JCT.

Our contributions: To our best knowledge, this is the first work that systematically studies: *how to schedule dependent coflows of multi-stage jobs, so that the total weighted job completion time can be minimized* (Section II).

We present a formal mathematical formulation. We also prove that this problem is strongly NP-hard. We relax it to a linear programming, so that lower bound can be obtained for performance evaluation (Section III).

Inspired by the optimal solution of the relaxed linear programming, we design an algorithm that runs in polynomial time to solve this problem with an approximation ratio of $(2M + 1)$, where M is the number of machines (Section IV).

Evaluation results demonstrate that, the largest gap between our algorithm and the lower bound is only 9.14%. We reduce the average JCT by up to 33.48% compared with Aalo, a heuristic multi-stage coflow scheduler. We reduce the total weighted JCT by up to 83.31% compared with LP-OV-LS, the state-of-the-art approximation algorithm of coflow scheduling, while our algorithm runs over 20× faster (Section V).

II. Related Work

Existing work, including heuristics and approximation algorithms, focus on scheduling coflows of single-stage jobs. The only exception is Aalo [8], which uses one small section to discuss a straight forward coflow heuristics to reduce the average JCT of multi-stage jobs.

Single-stage heuristics: Several work aim at developing heuristic coflow scheduling systems to minimize the average CCT. The coflow concept firstly appeared in Orchestra [4], which shows that even a simple FIFO discipline can significantly reduce the average CCT. The formal definition was presented later [5]. Varys uses the smallest-effective-bottleneck-first (SEBF) and minimum-allocation-for-desired-duration (MADD) heuristics to schedule coflows to minimize either the average CCT or deadline missing ratio [6]. Barrat exploits multiplexing to prevent head-of-line blocking to small coflows [7]. Stream aims at decentralized coflow scheduling [10]. These algorithms do not consider dependent relationships among coflows of multi-stage jobs. Explicitly handling dependency, our coflow scheduling algorithm has a bounded approximation ratio for multi-stage jobs.

Single-stage approximation algorithms: There are also some theoretical works aim at minimizing the total weighted CCT with approximation algorithms. The first polynomial-time deterministic approximation algorithm has an approximation ratio of $\frac{67}{3}$ by relaxing the problem to a time-indexed linear programming [11]. Khuller *et al.* proposed a 12-approximation algorithm by building a bridge towards concurrent open shop problem [12]. Luo *et al.* announced a 2-approximation algorithm [13]. Unfortunately, it has an incorrect relaxation and has been proven inaccurate later by a quite simple counter-example [14]. Recently, Shafiee *et al.* proposed a 5-approximation algorithm by relaxing the problem to a linear programming [14], [15]. Again, all these algorithms focus on minimizing CCT while ignore its divergence to JCT in the context of multi-stage jobs.

TABLE I
Notations of constants

Symbol	Definition
N	the number of jobs
K	the total number of coflows
M	the number of machines
$\mathbb{N} = \{1, 2, \cdots, N\}$	the job set
$\mathbb{K} = \{1, 2, \cdots, K\}$	the coflow set
$\mathbb{M} = \{1, 2, \cdots, M\}$	the machine set
$\mathbb{J}_n \in 2^{\mathbb{K}}, n \in \mathbb{N}$	the n-th job
$w_n, n \in \mathbb{N}$	the weight of the n-th job
$r_n, n \in \mathbb{N}$	the release time of the n-th job
$f_{ij}^k, k \in \mathbb{K}, i, j \in \mathbb{M}$	the flow $(i \to j)$ in coflow k
$p_{ij}^k, k \in \mathbb{K}, i, j \in \mathbb{M}$	the total bytes of flow f_{ij}^k

Multi-stage heuristics: Aalo developed a local queueing system in sender ends with the heuristics of discretized coflow-aware least-attained service (D-CLAS) to minimize the average CCT. To handle multi-stage scenarios, the simple heuristics is to prioritize coflows based on their dependency orders [8]. It has neither formal formulation nor analysis. Minimize the average JCT is just a special case of minimize the total weighted JCT, where all jobs have an equal weight.

III. Formulation and Analysis

We first present the original formulation of multi-stage coflow scheduling problem and prove its strong NP-Hardness. We relax it to a linear programming, which is essential for the construction of our approximation algorithm.

A. Settings

We abstract the network topology as a non-blocking big switch with M ports, each of which connects with a machine via a link of unit bandwidth; coflow property are known as a priori, such as the source, destination and bytes of each flow in it, just as what prior works do [6], [8], [11], [12], [15]. Recent advances in datacenter fabrics [16], [17] make it practical. Link-sharing and preemptions are allowed. We focus on network scheduling thus the computation duration of each reducer is ignored in the mathematical analysis. We use this abstraction to simplify our analysis; we do not require or enforce this in our evaluation.

B. Original Formulation

The multi-stage coflow scheduling problem can be formulated as follows, and the notations we used are shown in Table I and Table II.

TABLE II
NOTATIONS OF VARIABLES

Symbol	Definition
J_n	the completion time of the n-th job
C_k	the completion time of the k-th coflow
F_{ij}^k	the completion time of flow f_{ij}^k
$f_{ij}^k(t)$	the instantaneous transmission rate of flow f_{ij}^k at time t

$$\textbf{(O)}\ \min \sum_{n=1}^{N} w_n J_n \tag{1}$$

$$\text{s.t. } J_n = \max_{k\in\mathbb{J}_n} C_k, \forall n \in \mathbb{N}; \tag{2}$$

$$C_k = \max_{i,j\in\mathbb{M}} F_{ij}^k, \forall k \in \mathbb{K}; \tag{3}$$

$$\int_{r_n}^{F_{ij}^k} f_{ij}^k(t)dt = p_{ij}^k, \forall i,j \in \mathbb{M}, n \in \mathbb{N}, k \in \mathbb{J}_n; \tag{4}$$

$$\int_{0}^{r_n} f_{ij}^k(t)dt = 0, \forall i,j \in \mathbb{M}, n \in \mathbb{N}, k \in \mathbb{J}_n; \tag{5}$$

$$\int_{0}^{C_{k'}} f_{ij}^k(t)dt = 0, \forall i,j \in \mathbb{M}, n \in \mathbb{N}, k' \prec k \in \mathbb{J}_n; \tag{6}$$

$$\sum_{j\in\mathbb{M}}\sum_{k\in\mathbb{K}} f_{ij}^k(t) \le 1, \forall i \in \mathbb{M}, t \in [0,T]; \tag{7}$$

$$\sum_{i\in\mathbb{M}}\sum_{k\in\mathbb{K}} f_{ij}^k(t) \le 1, \forall j \in \mathbb{M}, t \in [0,T]; \tag{8}$$

$$f_{ij}^k(t) \ge 0, \forall i,j \in \mathbb{M}, n \in \mathbb{N}, k \in \mathbb{J}_n, t \in [0,T]. \tag{9}$$

Eq. (1) is the objective of our problem, *i.e.*, minimizing the total weighted JCT, while all of the others are constraints. JCT and CCT are defined by Eq. (2)(3), respectively. Eq. (4) guarantees that all bytes of each flow must be transmitted within proper time intervals and gives the definition of flow completion time in addition. Eq. (5) describes the constraints of release time, that is to say, a flow is allowed to transmit bytes only after the job it belongs to has been released. Eq. (6) describes the precedence constraints to guarantee that a flow cannot transmit any byte before all of its dependent coflows have finished. Eq. (7)(8) guarantee that for each port, the total data rate cannot exceed the port capacity (normalized to 1) at each time. Finally, Eq. (9) guarantees a non-negative data rate for each flow, which implies that preemptions are allowed (*i.e.*, the event that data rate turns to 0 is indeed a preemption).

Theorem III.1. *(O) is strongly NP-Hard.*

Proof. We will prove it by reducing **(O)** to the problem of coflow scheduling to minimize total weighted coflow completion time, which has proven to be strongly NP-Hard [11]. Given an arbitrary instance of Coflow Scheduling Problem (CSP), we construct a corresponding instance of **(O)** as following: supposing there are n coflows in CSP with release time r_i and weight w_i, we construct n jobs and each job has exact one coflow; for each job, the release time and weight are just those of the coflow in it. Thus, the solution of the constructed instance is exactly the solution of the given CSP instance. Therefore, if we could solve the constructed instance in polynomial time, we can also solve the CSP instance in polynomial time, which implies **(O)** is strongly NP-Hard even there is only one coflow in each job and no coflow dependency exists. □

C. Relaxed Formulation

Note that the original formulation **(O)** is a complicated nonlinear programming with infinite variables, which is hard to analyze and even harder to approximate. Thus we choose to relax **(O)** to an integer linear programming **(ILP)** firstly. We derive the constraints of **(ILP)** as following.

(1) Load Constraints. Denote $\mathbb{J} = \{\mathbb{J}_1, \mathbb{J}_2, \cdots, \mathbb{J}_N\}$ as the job set and it's clear that $\mathbb{J}$ is a partition of coflow set $\mathbb{K}$, which indicates that

$$\bigcup_{n\in\mathbb{N}} \mathbb{J}_n = \mathbb{K}; \tag{10}$$

$$\mathbb{J}_n \cap \mathbb{J}_{n'} = \emptyset, \forall n,n' \in \mathbb{N}, n \ne n'. \tag{11}$$

Thus Eq. (7) can be transformed into

$$\sum_{n'\in\mathbb{N}}\sum_{k\in\mathbb{J}_{n'}}\sum_{j\in\mathbb{M}} f_{ij}^k(t) \le 1. \tag{12}$$

For a specific Job n, let

$$\mathbb{N}_n \triangleq \{n' \in \mathbb{N} : J_{n'} \le J_n\} \tag{13}$$

and thus, Eq. (9)(12) indicate that

$$\sum_{n'\in\mathbb{N}_n}\sum_{k\in\mathbb{J}_{n'}}\sum_{j\in\mathbb{M}} f_{ij}^k(t) \le 1. \tag{14}$$

Now let's integrate the both sides of Eq. (14) from 0 to J_n and we can obtain that

$$\int_0^{J_n} \sum_{n'\in\mathbb{N}_n}\sum_{k\in\mathbb{J}_{n'}}\sum_{j\in\mathbb{M}} f_{ij}^k(t)dt \le J_n. \tag{15}$$

Swap the order between the integration and summations. By combining Eq. (4)(5)(6)(13), we can transform Eq. (15) into

$$\sum_{n'\in\mathbb{N}_n}\sum_{k\in\mathbb{J}_{n'}}\sum_{j\in\mathbb{M}} p_{ij}^k \le J_n. \tag{16}$$

Note that Eq. (16) is not a linear inequality, because the symbol $\mathbb{N}_n$ in the first summation is also a variable. Next we will introduce an order matrix $\boldsymbol{X} = [x_{ij}]$ to linearize the inequality. Let

$$x_{ij} \triangleq \mathbf{1}\{J_i < J_j\} \tag{17}$$

and ties between J_i and J_j are broken arbitrarily. $\mathbf{1}\{\cdot\}$ is the indicator function, which implies

$$x_{n'n} \in \{0,1\}, \forall n,n' \in \mathbb{N}, n \ne n'. \tag{18}$$

Note that the order matrix is indeed a representation of the strictly totally ordered relation among the job completion time, and the asymmetry and transitivity are equivalent to the following two linear constraints:

$$x_{nn'} + x_{n'n} = 1, \forall n,n' \in \mathbb{N}, n \ne n'; \tag{19}$$

$$x_{nn'} + x_{n'n''} \geq x_{nn''},$$
$$\forall n, n', n'' \in \mathbb{N}, n \neq n', n \neq n'', n' \neq n''. \quad (20)$$

Now Eq. (16) can be written as

$$\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k + \sum_{n':x_{n'n}=1}\sum_{k\in\mathbb{J}_{n'}}\sum_{j\in\mathbb{M}} p_{ij}^k \leq J_n, \quad (21)$$

which implies

$$\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k + \sum_{\substack{n'\in\mathbb{N}\\ n'\neq n}}\sum_{k\in\mathbb{J}_{n'}}\sum_{j\in\mathbb{M}} p_{ij}^k x_{n'n} \leq J_n,$$
$$\forall n \in \mathbb{N}, i \in \mathbb{M} \quad (22)$$

instantly due to the property of indicator function. Similarly, we have

$$\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}} p_{ij}^k + \sum_{\substack{n'\in\mathbb{N}\\ n'\neq n}}\sum_{k\in\mathbb{J}_{n'}}\sum_{i\in\mathbb{M}} p_{ij}^k x_{n'n} \leq J_n,$$
$$\forall n \in \mathbb{N}, j \in \mathbb{M}. \quad (23)$$

(2) Release Time Constraints. Eq. (7)(9) indicate that for a specific Job n, we have

$$\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} f_{ij}^k(t) \leq 1. \quad (24)$$

Integrating the both sides of Eq. (24) from r_n to J_n and finally, we have

$$J_n - r_n \geq \sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k \geq 0, \forall n \in \mathbb{N}. \quad (25)$$

Combining the derived constraints and properties, we obtain the relaxed formulation **(ILP)** as follow:

$$\textbf{(ILP)} \min \sum_{n=1}^{N} w_n J_n \quad (26)$$
$$\text{s.t. } (22)(23)(25)(19)(20)(18).$$

For integer linear programming is strongly NP-hard in general, we further relax **(ILP)** to a linear programming **(LP)**:

$$\textbf{(LP)} \min \sum_{n=1}^{N} w_n J_n \quad (27)$$
$$\text{s.t. } (22)(23)(25)(19)(20);$$
$$x_{n'n} \geq 0, \forall n, n' \in \mathbb{N}, n \neq n'. \quad (28)$$

Easy to see that **(LP)** is a relaxation of **(ILP)**. Denote the optimal solution of **(O)**, **(ILP)** and **(LP)** as OPT, OPT_{ILP} and OPT_{LP}, respectively. Due to the property of relaxation, we have

$$OPT_{LP} \leq OPT_{ILP} \leq OPT \quad (29)$$

for a minimization problem, which concludes that OPT_{LP} is a lower bound of the optimal solution of **(O)**.

IV. Approximation Algorithm

In general, the coflow scheduling problem can be reduced to a corresponding concurrent open shop problem [11], [18]. Multi-stage coflow scheduling problem is close to the $PDm|r_j, pmpt, prec|\sum w_j C_j$ problem[1], but few work aimed at this problem in the past. To the best of our knowledge, there are only approximation algorithms for this problem dealing with quite special cases, *e.g.*, the cases with at most two dependency chains [21]. Moreover, our problem is much knottier for two reasons: (1) Coflow scheduling is more complicated inherently due to coupled resource constraints [6]; (2) tree dependencies (even DAG dependencies) are common in current data parallel frameworks [5], [8], thus aiming at special type of dependencies is meaningless.

[1]Represented in improved $\alpha|\beta|\gamma$ notation [19], [20].

Algorithm 1: MCS algorithm

```
Solve the linear programming (LP) and denote the job
  completion time in the optimal solution as J̃_n, n ∈ ℕ.
Sort and reindex all jobs such that
                J̃_1 ≤ J̃_2 ≤ ··· J̃_N.                (30)
repeat
    call Update when a job released.
until all jobs finished

function Update
    Suspend all active coflows.
    List all released but not finished coflows in table L.
    Sort the coflows in L with a topological-sorting
      algorithm according to their dependencies.
    Sort the coflows in L with an arbitrary stable
      sorting algorithm (e.g., merge sorting) according
      to the jobs they belong to in the order of Eq. (30).
    for i = 1 → |L| do
        Decompose coflow L_i into k slices with
          Algorithm 2 and transmit them one by one
          with backfilling.
```

In this section, we proposed an event-driven approximation algorithm for Multi-stage Coflow Scheduling, namely MCS. To make it clear, we use matrix $\boldsymbol{C} = [c_{ij}] \in \mathcal{R}^{M\times M}$ to represent a coflow, where c_{ij} is the transmitted bytes from machine i to machine j in this coflow. Correspondingly, we denote $||\boldsymbol{C}|| = \max\{||\boldsymbol{C}||_1, ||\boldsymbol{C}||_\infty\}$ as the bottleneck bytes of coflow $\boldsymbol{C}$, where $||\cdot||_p$ is the p-norm.

A. Algorithm Design

To approximately solve **(O)** in polynomial time, we designed an event-based algorithm MCS, shown as Algorithm 1. At the beginning, we solve the linear programming **(LP)**, then sort and reindex all jobs according to their completion time in **(LP)**. For each job, the new index is regarded as its transmission priority, and a job with smaller index has a higher priority. The priority of a coflow is the priority the job it belongs to.

When a new job is released, all of the active jobs will be suspended and rescheduled. Specifically, we first sort all of the active coflows according to their priority, ties are broken according to their dependencies. Next, we schedule the coflows one by one, and each coflow is scheduled with our proposed

Algorithm 2: IBvN decomposition algorithm

Input: nonnegative matrix $\tilde{\boldsymbol{A}} = [a_{ij}] \in \mathcal{R}^{n\times n}$
Output: weights $\tilde{c}_l$, incomplete permutation matrices $\tilde{\boldsymbol{P}}_l = [p_{ij}^{(l)}] \in \mathcal{R}^{n\times n}$, $l = 1, 2, \cdots, k'$

1 Augment $\tilde{\boldsymbol{A}}$ to $||\tilde{\boldsymbol{A}}||\boldsymbol{A}$, where $\boldsymbol{A}$ is a doubly stochastic matrix. // see [11]
2 Decompose $\boldsymbol{A}$ into permutation matrices such that $\boldsymbol{A} = \sum_{i=1}^{k} c_i \boldsymbol{P}_i$. // see [23]
3 **for** $i = 1 \to n$ **do**
4 **for** $j = 1 \to n$ **do**
5 Find the index m such that $\sum_{l=1}^{m-1} c_l p_{ij}^{(l)} < a_{ij} \le \sum_{l=1}^{m} c_l p_{ij}^{(l)}$.
6 For all $l > m$, $p_{ij}^{(l)} \leftarrow 0$.
7 **if** $a_{ij} < \sum_{l=1}^{m} c_l p_{ij}^{(l)}$ **then**
8 Transform $c_m \boldsymbol{P}_m$ into $c_{m1}\boldsymbol{P}_{m1} + c_{m2}\boldsymbol{P}_{m2}$, where $c_{m2} = \sum_{l=1}^{m} c_l p_{ij}^{(l)} - a_{ij}$ and $c_{m1} = c_m - c_{m2}$, $\boldsymbol{P}_{m1} = \boldsymbol{P}_{m2} = \boldsymbol{P}_m$.
9 $p_{ij}^{(m2)} \leftarrow 0$.
10 Rearrange all weights and all incomplete permutation matrices such that $\tilde{\boldsymbol{A}} = \sum_{i=1}^{k'} c_i \tilde{\boldsymbol{P}}_i$.

Incomplete Birkhoff-von Neumann (IBvN) decomposition algorithm, inspired by the famous Birkhoff-von Neumann (BvN) theorem:

Theorem IV.1. [22] (BvN theorem) *Doubly stochastic matrix $\boldsymbol{A} \in \mathcal{R}^{n\times n}$ can be decomposed as $\boldsymbol{A} = \sum_{i=1}^{k} c_i \boldsymbol{P}_i$, where $c_i \in (0, 1)$ and $\boldsymbol{P}_i$ is a permutation matrix for each i, $\sum_{i=1}^{k} c_i = 1$, $k \le n^2 - 2n + 2$.*

For a specific coflow $\boldsymbol{C}$, if $\frac{\boldsymbol{C}}{||\boldsymbol{C}||}$ is a doubly stochastic matrix, *i.e.*, the bytes are uniformly distributed in each ingress and egress port, we may decompose it into k weighted permutation matrix and scheduling them directly. Two polynomial decomposition algorithms are given by [23]. Under this condition, there is no link-sharing thus no congestion, and each link can be fully-used.

However, in most cases, $\frac{\boldsymbol{C}}{||\boldsymbol{C}||}$ is far away from a doubly stochastic matrix. To deal with this situation, we propose the IBvN decomposition algorithm to decompose a nonnegative matrix into k weighted incomplete permutation matrix[2] , shown as Algorithm 2. The following theorem guarantees the feasibility and polynomial time of this algorithm.

Theorem IV.2. *Nonnegative matrix $\tilde{\boldsymbol{A}} \in \mathcal{R}^{n\times n}$ can be decomposed as $\tilde{\boldsymbol{A}} = \sum_{i=1}^{k'} \bar{c}_i \tilde{\boldsymbol{P}}_i$, where $\bar{c}_i \in (0, ||\tilde{\boldsymbol{A}}||)$ and $\tilde{\boldsymbol{P}}_i$ is an incomplete permutation matrix for each i, $\sum_{i=1}^{k'} \bar{c}_i = ||\tilde{\boldsymbol{A}}||$, $k' \le 2n^2 - 2n + 1$.*

[2]We define a binary matrix $\boldsymbol{P} = [p_{ij}] \in \mathcal{R}^{n\times n}$ as an incomplete permutation matrix if there is at most one entry of 1 in each row and each column.

Proof. By [11], nonnegative matrix $\tilde{\boldsymbol{A}}$ can be augmented to matrix $||\tilde{\boldsymbol{A}}||\boldsymbol{A}$, where $\boldsymbol{A}$ is a doubly stochastic matrix. According to Theorem IV.1, $\boldsymbol{A}$ can be decomposed into k permutation matrices $\boldsymbol{P}_1, \cdots, \boldsymbol{P}_k$ with corresponding weights $c_1, \cdots, c_k \in (0, 1)$ such that $\sum_{i=1}^{k} c_i = 1$, which implies that $\tilde{c}_1, \cdots, \tilde{c}_{k'} \in (0, 1)$ and $\sum_{i=1}^{k'} \tilde{c}_i = 1$. Noting that $||\tilde{\boldsymbol{A}}||\boldsymbol{A}$ is augmented from $\tilde{\boldsymbol{A}}$, we have $\bar{c}_i = ||\boldsymbol{A}||\tilde{c}_i$ for each i, which concludes that $\bar{c}_i \in (0, ||\tilde{\boldsymbol{A}}||)$ and $\sum_{i=1}^{k'} \bar{c}_i = ||\tilde{\boldsymbol{A}}||$. It's clear that Line 8 in Algorithm 2 can be executed by at most $n^2 - 1$ times, which generates at most $n^2 - 1$ new incomplete permutation matrices. As a result, $k' \le (n^2 - 2n + 2) + (n^2 - 1) = 2n^2 - 2n + 1$, which completes the proof. □

Theorem IV.3. *MCS algorithm runs in polynomial time.*

Proof. We first prove the following lemma.

Lemma IV.1. *IBvN decomposition algorithm runs in polynomial time.*

Proof. We omit the proof due to space limitation. See [11], [23] for reference. □

For the preprocessing phase of MCS algorithm, the linear programming (Line 1 in Algorithm 1) can be solved in polynomial time using ellipsoid algorithm [24] or projective algorithm [25], while sorting and reindexing (Line 2 in Algorithm 1) run in the time of $\mathcal{O}(n \log n)$.

When a job is released, the Update function is called. By digging into the Update function, we can see that Line 8-11 in Algorithm 1 runs in polynomial time, while from Lemma IV.1, we know that the IBvN decomposition algorithm (Line 13 in Algorithm 1) runs in polynomial time, and so does the Update function. Note that the MCS algorithm runs in polynomial time. This completes the proof. □

B. Performance Analysis

The following theorem indicates the performance of our proposed algorithm.

Theorem IV.4. *MCS is a $(2M + 1)$-approximation algorithm, where M is the number of machines.*

Proof. First let's derive the lower bound of the optimal solution, *i.e.*, OPT_{LP}. Note that OPT_{LP} is the summation of weighted job completion time, thus we investigate the lower bound of the job completion time.

Lemma IV.2. *In (LP), the job completion time of the l-th job $\tilde{J}_l \ge \frac{1}{2M} \sum_{n=1}^{l} \sum_{k\in\mathbb{J}_n} \sum_{i\in\mathbb{M}} \sum_{j\in\mathbb{M}} p_{ij}^{k}$.*

Proof. For simplicity, we define

$$\mu_{in} = \sum_{k\in\mathbb{J}_n} \sum_{j\in\mathbb{M}} p_{ij}^{k}, \forall i \in \mathbb{M}, n \in \mathbb{N}. \tag{31}$$

Thus, Eq. (22) can be written as

$$\mu_{in} + \sum_{\substack{n'\in\mathbb{N} \\ n'\neq n}} \mu_{in'} x_{n'n} \le \tilde{J}_n, \tag{32}$$

which indicates that

$$\mu_{in}^2 + \sum_{\substack{n' \in \mathbb{N} \\ n' \neq n}} \mu_{in'}\mu_{in}x_{n'n} \leq \mu_{in}\tilde{J}_n, \tag{33}$$

and thus,

$$\begin{aligned}\sum_{n=1}^{l}\mu_{in}^2 + \sum_{n=1}^{l}\sum_{\substack{n' \in \mathbb{N} \\ n' \neq n}} \mu_{in'}\mu_{in}x_{n'n} &\leq \sum_{n=1}^{l}\mu_{in}\tilde{J}_n \\ &\leq \left(\sum_{n=1}^{l}\mu_{in}\right)\tilde{J}_l,\end{aligned} \tag{34}$$

where the last inequality comes from Eq. (30). Note that for any $n \leq l$ and $n' > l$, we have $x_{n'n} = 0$, thus

$$\begin{aligned}\sum_{n=1}^{l}\sum_{\substack{n' \in \mathbb{N} \\ n' \neq n}} \mu_{in'}\mu_{in}x_{n'n} &= \sum_{n=1}^{l}\sum_{\substack{n'=1 \\ n' \neq n}}^{l} \mu_{in'}\mu_{in}x_{n'n} \\ &= \frac{1}{2}\left(\sum_{n=1}^{l}\mu_{in}\right)^2 - \frac{1}{2}\sum_{n=1}^{l}\mu_{in}^2.\end{aligned} \tag{35}$$

Combining Eq. (34)(35), we have

$$\frac{1}{2}\sum_{n=1}^{l}\mu_{in}^2 + \frac{1}{2}\left(\sum_{n=1}^{l}\mu_{in}\right)^2 \leq \left(\sum_{n=1}^{l}\mu_{in}\right)\tilde{J}_l, \tag{36}$$

which indicates that

$$\tilde{J}_l \geq \frac{1}{2}\sum_{n=1}^{l}\mu_{in} = \frac{1}{2}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k, \forall l \in \mathbb{N}, i \in \mathbb{M}. \tag{37}$$

Similarly, we have

$$\tilde{J}_l \geq \frac{1}{2}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}} p_{ij}^k, \forall l \in \mathbb{N}, j \in \mathbb{M}. \tag{38}$$

Combining Eq. (37)(38), we have

$$\begin{aligned}\tilde{J}_l &\geq \frac{1}{2}\max\left\{\max_{i\in\mathbb{M}}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k, \max_{j\in\mathbb{M}}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}} p_{ij}^k\right\} \\ &\geq \frac{1}{2M}\max\left\{\sum_{i\in\mathbb{M}}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k, \sum_{j\in\mathbb{M}}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}} p_{ij}^k\right\} \\ &= \frac{1}{2M}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}}\sum_{j\in\mathbb{M}} p_{ij}^k,\end{aligned} \tag{39}$$

which completes the proof. □

Next, we investigate the job completion time in our approximation algorithm.

Lemma IV.3. *In MCS algorithm, the job completion time of the l-th job $J_l \leq (2M+1)\tilde{J}_l$.*

Proof. Let's consider an easier case for which all jobs release at the same time, in another word, for all $l \in \mathbb{N}, r_l = 0$. The total idle time caused by precedence constraints and the total actual transmission time for all of the jobs $l' < l$ compose the job completion time J_l. Thus

$$\begin{aligned}J_l &\leq \sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\max\left\{\max_{i\in\mathbb{M}}\sum_{j\in\mathbb{M}} p_{ij}^k, \max_{j\in\mathbb{M}}\sum_{i\in\mathbb{M}} p_{ij}^k\right\} \\ &\leq \sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\max\left\{\sum_{i\in\mathbb{M}}\sum_{j\in\mathbb{M}} p_{ij}^k, \sum_{j\in\mathbb{M}}\sum_{i\in\mathbb{M}} p_{ij}^k\right\} \\ &= \sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}}\sum_{j\in\mathbb{M}} p_{ij}^k \leq 2M\tilde{J}_l.\end{aligned} \tag{40}$$

Note the first inequality in Eq. (40) is tight and the equality holds when all of the coflows in each job have strict linear dependencies, while the last inequality comes from Lemma IV.2. When jobs have arbitrary release time, Eq. (40) can be written as

$$J_l \leq r_l + 2M\tilde{J}_l, \tag{41}$$

because the idle time introduced by release time can be upper-bounded by r_n. Taking Eq. (25) into consideration, we have

$$J_l \leq \tilde{J}_l + 2M\tilde{J}_l = (2M+1)\tilde{J}_l, \tag{42}$$

which completes the proof. □

From Lemma IV.3, the solution of our approximation algorithm can be represented as

$$\begin{aligned}SOL &= \sum_{l=1}^{N} w_l J_l \leq (2M+1)\sum_{l=1}^{N} w_l\tilde{J}_l \\ &= (2M+1)OPT_{LP}\end{aligned} \tag{43}$$

and thus the approximation ratio

$$\alpha = \frac{SOL}{OPT_{LP}} \leq 2M+1, \tag{44}$$

which completes the proof. □

However, this bound is not tight, because for the last inequalities in Eq. (39)(40), equalities can never hold simultaneously due to the property of *max* operator. It seems an unavoidable problem, because the completion time can be hardly bounded by their bytes directly when dependencies exist.

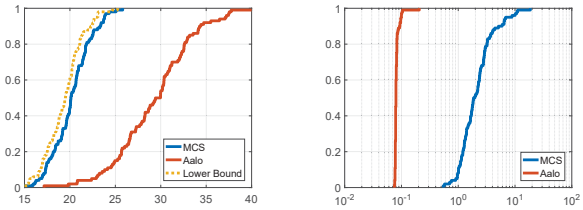
C. Extension Results

There are two extensions of our algorithm. In special cases, our algorithm has a constant approximation ratio.

Corollary IV.1. *When bytes are uniformly distributed in each ingress and egress port for each coflow, MCS is a 3-approximation algorithm.*

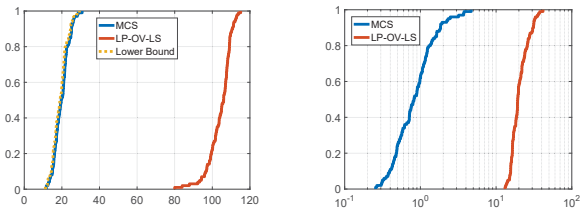
Proof. When bytes are uniformly distributed in each ingress and egress port for each coflow, *i.e.*,

$$\sum_{j\in\mathbb{M}} p_{ij}^k = \sum_{i\in\mathbb{M}} p_{ij}^k = c_k, \forall i, j \in \mathbb{M}, k \in \mathbb{K}, \tag{45}$$



(a) Normalized average JCT (b) Actual execution time (s)

Fig. 2. CDF of normalized average JCT and actual execution time



(a) Normalized total weighted JCT (b) Actual execution time (s)

Fig. 4. CDF of normalized total weighted JCT and actual execution time

Eq. (39) can be transformed into

$$\tilde{J}_l^* \geq \frac{1}{2} \sum_{n=1}^{l} \sum_{k \in \mathbb{J}_n} c_k, \tag{46}$$

while Eq. (42) can be transformed into

$$J_l^* \leq r_l + \sum_{n=1}^{l} \sum_{k \in \mathbb{J}_n} c_k \leq \tilde{J}_l^* + 2\tilde{J}_l^* = 3\tilde{J}_l^*. \tag{47}$$

Thus

$$\alpha^* = \frac{SOL^*}{OPT_{LP}^*} = \frac{\sum_{l=1}^{N} w_l J_l^*}{\sum_{l=1}^{N} w_l \tilde{J}_l^*} \leq 3, \tag{48}$$

which completes the proof. □

Corollary IV.2. *When all of the jobs have the same release time, MCS is a $2M$-approximation algorithm in general cases, and is a 2-approximation algorithm when bytes are uniformly distributed in each ingress and egress port for each coflow.*

Proof. Easy to see by setting $r_l = 0$ for all $l \in \mathbb{N}$, without loss of generality. □

V. Evaluation

A. Methodology

Simulator: We evaluate our algorithm with an event-based flow simulator by performing a replay of the collected Facebook logs, which are widely accepted as a benchmark in both system works and theoretical works [6], [8], [11], [15], [26]. For there is no prior work aimed at multi-stage coflow scheduling to minimizing the total weighted JCT, we validated our algorithm by comparing with two closest algorithms: Aalo and LP-OV-LS. Aalo is the only algorithm that considers multi-stage coflow scheduling, however, it cannot handle the weighted scenarios and has no performance guarantees [8]. LP-OV-LS is the state-of-the-art approximation algorithm for coflow scheduling to minimize the total weighted CCT, however, it can hardly face with multi-stage scenarios [14], [15]. We enable work conservation for all algorithms to guarantee the fairness.

Workload: The coflow information in Facebook logs is incomplete. For each coflow, the Facebook logs contain its sender machines, receiver machines, and transmitted bytes in receiver level, instead of in flow level, thus we partition the bytes in each receiver to each sender pseudo-uniformly with a small popple to generate flows. Besides, the Facebook logs contain only coflow information instead of job information, thus we randomly partition these coflows into some jobs such that each job contains α coflows in expectation, and DAG dependencies among coflows are randomly generated; the release time of the jobs follows a Poisson process with parameter θ; the weights of the jobs follow a uniform distribution, and are normalized such that all of the weights sum up to 1.

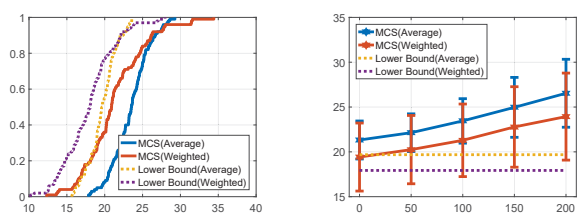
Metrics: We evaluate these algorithms with two metrics by default: the total weighted JCT and actual running time of these algorithms. The lower bound of our algorithm is also taken into consideration by solving **(LP)**. All of the data points are collected from 100 runs with random workload, as is aforementioned.

Summary: We summarize our experimental results as answers to the following questions.

- **How close is MCS to its lower bound?** We evaluate our algorithm in both weighted and non-weighted scenarios, and the largest gap between our algorithm and LP lower bound is 9.14%, which implies that our algorithm finds a quite good solution in practice.
- **Does MCS perform better that state-of-the-art algorithms?** For multi-stage coflow scheduling, we compare our algorithm with the only existing (heuristic) algorithm, and our algorithm reduces the average JCT by up to 33.48%; for coflow scheduling, we compare our algorithm with the state-of-the-art approximation algorithm, and our algorithm reduces the average JCT by up to 83.31%, while our algorithm runs over 20× faster.
- **How does MCS perform in a large parameter space?** We further investigate the influence of the number of machines (M), the average interval of job arrival (θ) and the average number of coflows in each job (α). Results show that our algorithm works well consistently.

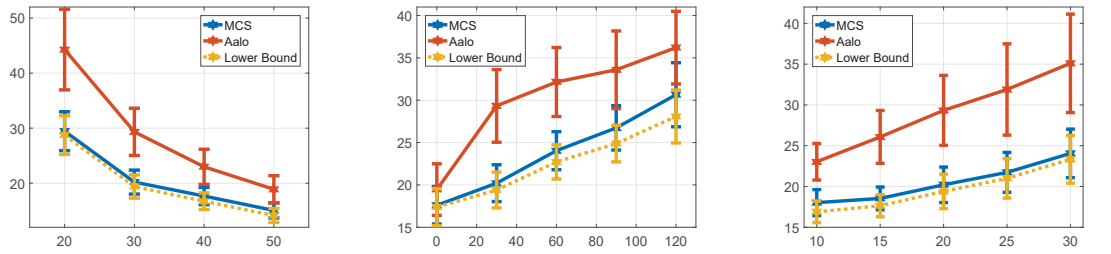
B. Special Cases

In order to compare MCS with Aalo, we first evaluate our algorithm in special cases, *i.e.*, all jobs have the same weights, namely non-weighted scenarios. In this case, the optimization objective is indeed equivalent to minimizing the average JCT.



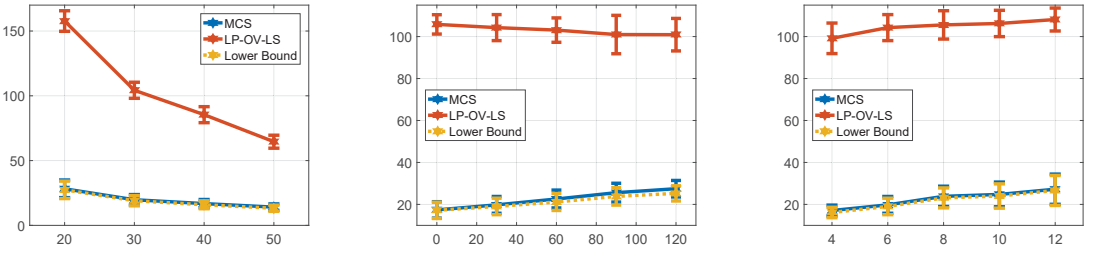
(a) CDF of JCT (b) Influence of comp. delay (s)

Fig. 6. Influence of average coflow computation delay



(a) The # of machines (M) (b) Average interval of job arrival (θ) (c) Average # of coflows in each job (α)

Fig. 3. The normalized average JCT with different parameters



(a) The # of machines (M) (b) Average interval of job arrival (θ) (c) Average # of coflows in each job (α)

Fig. 5. The normalized total weighted JCT with different parameters

We choose $M = 30, \theta = 30, \alpha = 20$ as the default parameters, and results are shown in Fig. 2 and Fig. 3.

As illustrated, all of the jobs are randomly combined by the coflows in Facebook logs with random dependencies, while their release time is randomly generated, thus it's necessary to investigate the cumulative distribution function (CDF) of the average JCT and the time spent on scheduling. As shown in Fig. 2(a), our algorithm has a much stabler performance than Aalo: for our algorithm, the average JCT varies from about 15 to 26, with coefficient of variation $CV_{MCS} = 0.1077$; for Aalo, it varies from about 17 to 40, and the coefficient of variation $CV_{Aalo} = 0.1462$. However, our algorithm is slower than Aalo. As a heuristic algorithm, Aalo schedules coflows with simple rules rather than complex computation, thus its execution time concentrates on about 0.08 s; by contract, our algorithm has to solve a linear programming in preprocessing phase, which takes more than 95% of the execution time, as a result, our algorithm needs about 2 s on average.

Next, we investigate the influence of the number of machines, shown in Fig. 3(a). Note that the total bandwidth grows linearly as the number of machines growing, thus for both the algorithms, the average JCT decreases. It's clear that the average JCT is not inversely proportional to the number of machines, due to the non-uniformity of coflow size, time distribution and space distribution. Compared with Aalo, we reduce the average JCT by up to 33.48%, and the largest gap between our algorithm and its lower bound is only 6.44%.

Then we investigate the influence of the average interval of job arrival, *i.e.* the parameter θ in Poisson process, shown in Fig. 3(b). In fact, the link load is related to this parameter. When all coflows arrived at the same time, *i.e.*, $\theta = 0$, the link is always fully used for a long time; when θ becomes larger and larger, finally there will be at most only one active coflow at any time. Exactly, this is the reason why the gap between our algorithm and Aalo is firstly larger and then becomes smaller: when link load becomes too light or too heavy, the scheduling algorithms usually play a quite small role. Compared with Aalo, we reduce the average JCT by up to 31.10%, and the largest gap between our algorithm and its lower bound is 9.14%.

Finally, let's investigate the influence of the job structure, *i.e.*, the average number of coflows in each job. From Fig. 3(c), we can see that all of three curves increase linearly with similar slopes. This is because, when a job contains more coflows, in general, it has more bytes to transmit, thus the JCT becomes larger. Compared with Aalo, we reduce the average JCT by up to 31.87%, and the largest gap between our algorithm and its lower bound is only 5.07%.

C. General Cases

Now we evaluate our algorithm in general cases, *i.e.*, each job has a random weight, and choose the state-of-the-art approximation algorithm of coflow scheduling LP-OV-LS for comparison. Note that the time complexity of LP-OV-LS is related with the number of coflows, instead of the number of jobs, thus we have to reduce the total number of coflows to guarantee that LP-OV-LS can be finished in reasonable time. We choose $M = 30, \theta = 30, \alpha = 6$ as the default parameters, and results are shown in Fig. 4 and Fig. 5.

Let's investigate the CDF of the total weighted JCT and the time spent on scheduling. As shown in Fig. 4(a), the total weighted JCT of LP-OV-LS is over 5× larger than that of our algorithm due to improper optimization objective and the lack of consideration on coflow dependencies, which is essential for multi-stage coflow scheduling. Besides, LP-OV-LS runs over 20× slower than our algorithm, because it has to solve a linear programming with more variables and more constraints, as shown in Fig. 4(b).

Then we investigate the influence of aforementioned three parameters. As shown in Fig. 5(a), the total weighted JCT of both algorithms decreases as the number of machines growing. However, Fig. 5(b) is really confused that the total weighted JCT of LP-OV-LS should decreases as θ increasing. We think it's caused by improper optimization objective, because when we try to optimize CCT, the behavior of JCT is out of control in multi-stage scenarios. The influence of α in weighted scenarios is similar with that in non-weighted scenarios, shown in Fig. 5(c). Compared with LP-OV-LS, we reduce the average JCT by up to 83.31%, and the largest gap between our algorithm and its lower bound is 8.79%.

D. Further Evaluation

We further evaluate our algorithm in the scenarios when each coflow has a computation phase, as shown in Fig. 6. Firstly, we investigate the CDF of both the average JCT and total weighted JCT when each coflow has a 100 s computation delay on average, shown in Fig. 6(a). Then we evaluate our algorithm within a large range of computation delay from 0 s to 200 s, where the time cost on computation is comparable with that cost on network transmission. Results are shown in Fig. 6(b): as the time cost on computation growing, the average or total weighted JCT increases as expected. By comparing with Fig. 2(a) and Fig. 4(a), we can see that even each coflow has a 200 s computation delay on average, our algorithm still performs better than others. When the time cost on computation is far greater than that cost on network transmission, the gap among all algorithms becomes negligible, thus network scheduling comes to be meaningless.

VI. Conclusion

There are dependent relationships among coflows of multi-stage jobs in datacenters. As the first systematic work, we formulate coflow scheduling of multi-stage jobs as a problem to minimize the total weighted JCT. We design an approximation algorithm. Evaluation results show that our algorithm significantly outperforms state-of-the-art works.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments. This work was supported in part by the National Science and Technology Major Project of China under Grant Number 2017ZX03001013-003, the Fundamental Research Funds for the Central Universities under Grant Number 0202-14380037, the National Natural Science Foundation of China under Grant Numbers 61772265, 61602194, 61502229, 61672276, and 61321491, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

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

[2] "Apache hadoop." http://hadoop.apache.org.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[4] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM*, vol. 41, no. 4. ACM, 2011, pp. 98–109.

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

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

[7] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 431–442.

[8] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 393–406.

[9] "Tpc-ds." http://www.tpc.org/tpcds.

[10] H. Susanto, H. Jin, and K. Chen, "Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks," in *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.

[11] Z. Qiu, C. Stein, and Y. 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.

[12] S. Khuller and M. 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.

[13] 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 Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3366–3380, 2016.

[14] 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.

[15] ——, "Brief announcement: A new improved bound for coflow scheduling," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2017.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM 2009*.

[17] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Proc. ACM SIGDC 2015*. ACM, 2015, pp. 183–197.

[18] T. A. Roemer, "A note on the complexity of the concurrent open shop problem," *Journal of scheduling*, vol. 9, no. 4, pp. 389–396, 2006.

[19] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of discrete mathematics*, vol. 5, pp. 287–326, 1979.

[20] J. Y.-T. Leung, H. Li, and M. Pinedo, "Order scheduling in an environment with dedicated resources in parallel," *Journal of Scheduling*, vol. 8, no. 5, pp. 355–386, 2005.

[21] A. Agnetis, H. Kellerer, G. Nicosia, and A. Pacifici, "Parallel dedicated machines scheduling with chain precedence constraints," *European Journal of Operational Research*, vol. 221, no. 2, pp. 296–305, 2012.

[22] M. Marcus and R. Ree, "Diagonals of doubly stochastic matrices," *The Quarterly Journal of Mathematics*, vol. 10, no. 1, pp. 296–302, 1959.

[23] F. Dufossé and B. Uçar, "Notes on birkhoff–von neumann decomposition of doubly stochastic matrices," *Linear Algebra and its Applications*, vol. 497, pp. 108–115, 2016.

[24] L. G. Khachiyan, "A polynomial algorithm in linear programming," *Ussr Computational Mathematics & Mathematical Physics*, vol. 20, no. 80, pp. 1–3, 1979.

[25] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM, 1984, pp. 302–311.

[26] Y. Li, S. H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, and F. Lau, "Efficient online coflow routing and scheduling," in *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*. ACM, 2016, pp. 161–170.