Computer Networks 158 (2019) 193–205

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Scheduling dependent coflows to minimize the total weighted job completion time in datacenters☆

Bingchuan Tian, Chen Tian*, Bingquan Wang, Bo Li, Zehao He, Haipeng Dai, Kexin Liu, Wanchun Dou, Guihai Chen

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

ARTICLE INFO

Article history:
Received 3 March 2019
Accepted 14 May 2019
Available online 17 May 2019

Keywords:
Coflow scheduling
Approximation algorithm
Datacenter

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. 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 $(2M+1)$ in general case, and 3 in special case, where M is the number of hosts. Evaluation results demonstrate that, the largest gap between our algorithm and the lower bound is only 9.14%. In testbeds, we reduce the JCT by up to 81.65% comparing with pure DCTCP. In simulations, we reduce the average JCT by up to 33.48% comparing with Aalo, a heuristic multi-stage coflow scheduler; we reduce the total weighted JCT by up to 83.58% comparing with LP-OV-LS, the state-of-the-art approximation algorithm of coflow scheduling.

1. Introduction

Datacenter networks are critical to cloud computing. In modern datacenters, data-parallel frameworks (*e.g.*, MapReduce [2], Hadoop [3], Spark [4]) 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 [5].

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 [6]. Current work focuses 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 is only one coflow and time spent on computation can be considered as nearly constant [7–9].

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 stage with dependencies. For example, each TPC-DS query (of distributed database applications) is a directed-acyclic-graph (DAG) of multi-stage dataflow [10]. 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* [9]. *Starts-After* represents the existence of explicit

☆ This paper is an extended version of Tian et al. [1] in INFOCOM'18.

* Corresponding author.
E-mail addresses: bctian@smail.nju.edu.cn (B. Tian), tianchen@nju.edu.cn (C. Tian), 773854936@qq.com (B. Wang), libo_nju16@small.nju.edu.cn (B. Li), hezehao@qq.com (Z. He), haipengdai@nju.edu.cn (H. Dai), kxliu@small.nju.edu.cn (K. Liu), douwc@nju.edu.cn (W. Dou), gchen@nju.edu.cn (G. Chen).

https://doi.org/10.1016/j.comnet.2019.05.010
1389-1286/

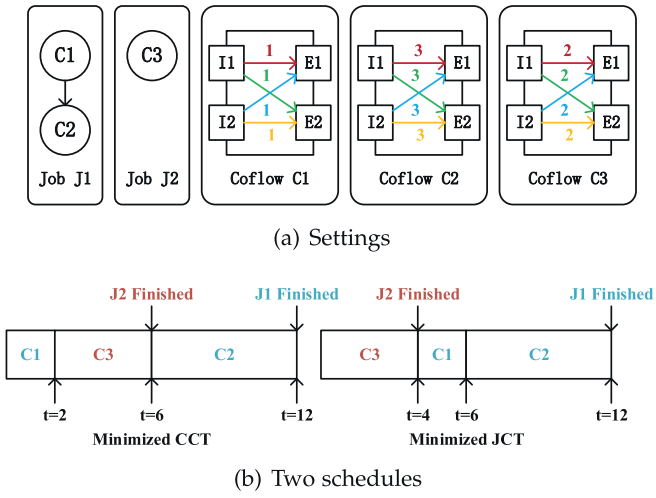


Fig. 1. A motivating example.

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 motivating example in Fig. 1. There are two equally-weighted 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 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 $\langle C_1, C_3, C_2\rangle$. The corresponding average JCT is $\frac{(2+4+6)+(2+4)}{2} = 9$. However, if we schedule the coflows in the order of $\langle C_3, C_1, C_2\rangle$, 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 2). We present a formal mathematical formulation. We relax it to a linear programming, so that lower bound can be obtained for performance evaluation (Section 3). 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 $(2M+1)$ for both non-oversubscribed and oversubscribed networks, where M is the number of hosts (Sections 4 and 5). With a special kind of coflow patterns, the approximation ratio can be reduced to 3. We design and implement a scheduling framework in our testbeds (Section 6). Evaluation results demonstrate that, the largest gap between our algorithm and the lower bound is only 9.14%. In testbeds, we reduce the average/total weighted JCT by up to 81.65%, comparing with pure DCTCP. In simulations, we reduce the average JCT by up to 33.48% comparing with Aalo, a heuristic multi-stage coflow scheduler. We reduce the total weighted JCT by up to 83.58% comparing with LP-OV-LS, a state-of-the-art approximation algorithm of coflow scheduling, while our algorithm runs over 20× faster (Section 7).

2. Related work

Existing work, including heuristics and approximation algorithms, focus on scheduling coflows of single-stage jobs. The only exception is Aalo [9], which uses a small section to discuss a straight forward heuristic to reduce the average JCT of multi-stage jobs.

Single-stage heuristics: Several work aims at developing heuristic coflow scheduling systems to minimize the average CCT. The coflow concept firstly appeared in Orchestra [5], which shows that even a simple FIFO discipline can significantly reduce the average CCT. The formal definition was presented later [6]. 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 [7]. Barrat exploits multiplexing to prevent head-of-line blocking to small coflows [8]. Stream aims at decentralized coflow scheduling [11]. 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 work aiming 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 [12]. Khuller *et al.* proposed a 12-approximation algorithm by building a bridge towards concurrent open shop problem [13]. Luo *et al.* announced a 2-approximation algorithm [14]. Unfortunately, it has been proven inaccurate later by a counter-example [15]. Recently, Shafiee *et al.* proposed a 5-approximation algorithm by relaxing the problem to a linear programming [15,16]. Again, all these algorithms focus on minimizing CCT while ignore its divergence to JCT in the context of multi-stage jobs.

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 heuristic is to prioritize coflows based on their dependency orders [9]. It has neither formal formulation nor analysis. Minimizing the average JCT is just a special case of minimizing the total weighted JCT, where all jobs have an equal weight.

3. Formulation and analysis

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

3.1. Settings

We abstract the network topology as a non-blocking big switch with M ports, each of which is connected with a host via a link of unit bandwidth; coflow properties are known as a priori, such as the source, destination and bytes of each flow in it, just as what prior work does [7,9,12,13,16]. Recent advances in datacenter fabrics [17,18] make it practical. Link-sharing and network preemptions are allowed.

For some jobs, whether a reducer's computation duration can be neglected depends on the speed of CPU and network: when

Table 1
Notations of constants.

Symbol	Definition
N	the number of jobs
K	the total number of coflows
M	the number of hosts
T	the maximum possible running time
$\mathbb{N} = \{1, 2, \cdots, N\}$	the job set
$\mathbb{K} = \{1, 2, \cdots, K\}$	the coflow set
$\mathbb{M} = \{1, 2, \cdots, M\}$	the host set
$\mathbb{J}_n \in 2^{\mathbb{K}}, n \in \mathbb{N}$	the nth job
$w_n, n \in \mathbb{N}$	the weight of the nth job
$r_n, n \in \mathbb{N}$	the release time of the nth 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

Table 2
Notations of variables.

Symbol	Definition
J_n	the completion time of the nth job
C_k	the completion time of the kth 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

CPU is faster, the received data can be processed at once without queuing, thus the reducer's completion time is indeed the network transmission time; when network is faster, received data that cannot be timely processed will be buffered, thus an extra computation phase is needed after network transmission. For simplicity, in mathematical analysis, we only focus on the first scenario and ignore the computation duration. We do not require or enforce this in our evaluation.

3.2. Original formulation

The multi-stage coflow scheduling problem can be formulated as follows. Notations of constant prior information are summarized in Table 1, while decision variables are summarized in Table 2.

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

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

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

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

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

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

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

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

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

$$T = \sum_{i \in \mathbb{M}} \sum_{j \in \mathbb{M}} \sum_{k \in \mathbb{K}} p_{ij}^k + \max_{n \in \mathbb{N}} r_n. \tag{10}$$

Eq. (1) is the objective of our problem, *i.e.*, minimizing the total weighted JCT; again, minimizing the average JCT is a special case of this objective. JCT and CCT are defined by Eqs. (2) and (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, which implies DAG dependencies among coflows in a job. Eqs. (7) and (8) guarantee that for each port (*i.e.*, host), the total data rate cannot exceed the port capacity (normalized to 1) at each time. Eq. (9) guarantees a non-negative data rate for each flow, which in addition implies that preemptions are allowed (*i.e.*, the event that data rate turns to 0 is indeed a preemption). Finally, Eq. (10) gives us an upper bound of the maximum possible running time T; though it is quite straightforward, it does not have any bad impact on the following work. **(O)** is NP-hard, which can be proved by reducing from the original coflow scheduling problem [7,12].

3.3. 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.

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

$$\bigcup_{n \in \mathbb{N}} \mathbb{J}_n = \mathbb{K}, \tag{11}$$

$$\mathbb{J}_n \cap \mathbb{J}_{n'} = \emptyset, \forall n, n' \in \mathbb{N}, n \ne n', \tag{12}$$

thus Eq. (7) can be transformed into

$$\sum_{n' \in \mathbb{N}} \sum_{k \in \mathbb{J}_{n'}} \sum_{j \in \mathbb{M}} f_{ij}^k(t) \le 1. \tag{13}$$

For a specific Job n, let

$$\mathbb{N}_n \triangleq \{n' \in \mathbb{N} : J_{n'} \le J_n\} \tag{14}$$

thus Eqs. (9) and (13) indicate that

$$\sum_{n' \in \mathbb{N}_n} \sum_{k \in \mathbb{J}_{n'}} \sum_{j \in \mathbb{M}} f_{ij}^k(t) \le 1. \tag{15}$$

Now let's integrate the both sides of Eq. (15) from 0 to J_n and we can obtain that

$$\int_0^{J_n} \sum_{n' \in \mathbb{N}_n} \sum_{k \in \mathbb{J}_{n'}} \sum_{j \in \mathbb{M}} f_{ij}^k(t)dt \le J_n. \tag{16}$$

Swap the order between the integration and summations. By combining Eqs. (4), (5), (6), (10) and (14), we can transform Eq. (16) into

$$\sum_{n' \in \mathbb{N}_n} \sum_{k \in \mathbb{J}_{n'}} \sum_{j \in \mathbb{M}} p_{ij}^k \le J_n. \tag{17}$$

Note that Eq. (17) 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

$$x_{ij} \triangleq \mathbf{1}\{J_i < J_j\} \tag{18}$$

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

$$x_{n'n} \in \{0, 1\}, \forall n, n' \in \mathbb{N}, n \ne n'. \tag{19}$$

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:

$$x_{nn'} + x_{n'n} = 1, \forall n, n' \in \mathbb{N}, n \neq n'; \tag{20}$$

$$x_{nn'} + x_{n'n''} \geq x_{nn''}, \quad \forall n, n', n'' \in \mathbb{N}, n \neq n', n \neq n'', n' \neq n''. \tag{21}$$

Now Eq. (17) can be written as

$$\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k + \sum_{n':x_{n'n}=1}\sum_{k\in\mathbb{J}_{n'}}\sum_{j\in\mathbb{M}} p_{ij}^k \leq J_n, \tag{22}$$

which implies

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

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

$$\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}} p_{ij}^k + \sum_{\substack{n'\in\mathbb{N}\\ n'\neq n}}\sum_{k\in\mathbb{J}_{n'}}\sum_{i\in\mathbb{M}} p_{ij}^k x_{n'n} \leq J_n, \quad \forall n \in \mathbb{N}, j \in \mathbb{M}. \tag{24}$$

Release-time Constraints: Eqs. (7) and (9) indicate that for a specific Job n, we have

$$\sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} f_{ij}^k(t) \leq 1. \tag{25}$$

Integrating the both sides of Eq. (25) from r_n to J_n and finally, we have

$$J_n - r_n \geq \sum_{k\in\mathbb{J}_n}\sum_{j\in\mathbb{M}} p_{ij}^k \geq 0, \forall n \in \mathbb{N}. \tag{26}$$

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

$$\textbf{(ILP)} \min \sum_{n=1}^{N} w_n J_n \tag{27}$$

s.t. (23)(24)(26)(20)(21)(19).

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

$$\textbf{(LP)} \min \sum_{n=1}^{N} w_n J_n \tag{28}$$

s.t. (23)(24)(26)(20)(21);

$$x_{n'n} \geq 0, \forall n, n' \in \mathbb{N}, n \neq n'. \tag{29}$$

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

$$OPT_{LP} \leq OPT_{ILP} \leq OPT \tag{30}$$

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

4. Approximation algorithms

In general, the coflow scheduling problem can be reduced to a corresponding concurrent open shop problem [12,19]. Multi-stage coflow scheduling problem is close to (however, not the same as) the $PDm|r_j, pmpt, prec|\Sigma w_j C_j$ problem, represented in improved $\alpha|\beta|\gamma$ notation [20,21], 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 [22]. Moreover, our problem is much knottier for two reasons: (1) Coflow scheduling is more complicated inherently due to coupled resource constraints [7]; (2) tree dependencies (even DAG dependencies) are common in current data-parallel frameworks [6,9], thus aiming at special types of dependencies is meaningless.

In this section, we proposed an event-driven approximation algorithm for Multi-stage Coflow Scheduling, namely MCS. The basic idea of MCS is to order coflows according to completion time in relaxed linear programming, which is used to solve and analysis all kinds of scheduling problems [15,16,23–27]. 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 host i to host 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.

4.1. 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)**. Here, total bytes of each flow, jobs' release time and jobs' weights are inputs, while the order matrix and jobs' completion time are decision variables (*i.e.*, the outputs of **(LP)**). Then, we sort and reindex all jobs according to their completion time in the optimal solution of **(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.

Algorithm 1: MCS algorithm.

Input: release time $r_{(\cdot)}$ and weight $w_{(\cdot)}$ of each job, total bytes of each flow $f_{(\cdot,\cdot)}^{(\cdot)}$

1 Solve the linear programming **(LP)** with above inputs and denote the job completion time in the optimal solution as $\tilde{J_n}$, $n \in \mathbb{N}$.
2 Sort and reindex all jobs such that
3 $\tilde{J_1} \leq \tilde{J_2} \leq \cdots \tilde{J_N}$. (31)
4 **repeat**
5 **call** Update **when** *a job released*.
6 **until** *all jobs finished*
7
8 **function** Update
9 Suspend all active coflows.
10 List all released but not finished coflows in table L.
11 Sort the coflows in L with a topological-sorting algorithm according to their dependencies.
12 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. (31).
13 **for** $i = 1 \rightarrow |L|$ **do**
14 Decompose coflow L_i into k coflow-slices with Algorithm 2 and transmit them one by one with backfilling.

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 priorities, ties are broken according to their dependencies. Next, we schedule the coflows one by one, and each coflow is scheduled with our proposed Incomplete

Birkhoff-von Neumann (IBvN) decomposition algorithm, inspired by the famous Birkhoff-von Neumann (BvN) theorem, which is fundamental to some network schedulers [12,28–30].

Theorem 4.1 ([31] 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*,* $\sum_{i=1}^{k} c_i = 1$*,* $k \le n^2 - 2n + 2$*.*

Definition 4.1 (Incomplete permutation matrix)**.** A binary matrix $\boldsymbol{P} = [p_{ij}] \in \mathcal{R}^{n\times n}$ is defined as an incomplete permutation matrix if there is at most one entry of 1 in each row and each column.

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 schedule them directly. Two polynomial decomposition algorithms are given by [32]. 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 (Algorithm 2) to decompose a nonnegative matrix into k weighted incomplete permutation matrix. As a result, a coflow is decomposed into some coflow-slices, and each slice can be directly transmitted without link-sharing and preemptions.

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 [12].
2 Decompose $\boldsymbol{A}$ into permutation matrices such that $\boldsymbol{A} = \sum_{i=1}^{k} c_i \boldsymbol{P}_i$ [32].
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 $\tilde{\boldsymbol{A}} = \sum_{i=1}^{k'} c_i \tilde{\boldsymbol{P}}_i$.

Note that the coflow-slice is represented as an incomplete permutation matrix instead of a permutation matrix, thus some links can be empty, which can harm network throughput. Here backfilling mechanism is used to keep network busy. Specifically, for a running coflow-slice (with highest priority), when the scheduler finds that some links are empty, the flow in other coflow-slices is allowed to run in advance if it can transmit data with nonzero rate and do not interfere with the current running coflow-slice. Flows in jobs with higher priority are considered firstly. The feasibility and polynomial running time of Algorithm 2 are described in following theorems:

Theorem 4.2. *Nonnegative matrix* $\tilde{\boldsymbol{A}} \in \mathcal{R}^{n\times n}$ *can be decomposed as* $\tilde{\boldsymbol{A}} = \sum_{i=1}^{k'} \tilde{c}_i \tilde{\boldsymbol{P}}_i$*, where* $\tilde{c}_i \in (0, ||\tilde{\boldsymbol{A}}||)$ *and* $\tilde{\boldsymbol{P}}_i$ *is an incomplete permutation matrix for each* i*,* $\sum_{i=1}^{k'} \tilde{c}_i = ||\tilde{\boldsymbol{A}}||$*,* $k' \le 2n^2 - 2n + 1$*.*

Proof. By [12], 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 4.1, $\boldsymbol{A}$ 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 4.3. *MCS algorithm runs in polynomial time.*

Proof. We first give the following lemma. Proof is trivial.

Lemma 4.1. *IBvN decomposition algorithm runs in polynomial time.*

For the preprocessing phase of MCS algorithm, the linear programming (Line 1 in Algorithm 1) can be solved in polynomial time using ellipsoid algorithm [33] or projective algorithm [34], 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 4.1, 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. □

4.2. Performance analysis

The following theorem indicates the performance of our proposed algorithm.

Theorem 4.4. *MCS is a* $(2M + 1)$*-approximation algorithm, where* M *is the number of hosts.*

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 4.2. *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

$$\mu_{in} = \sum_{k\in\mathbb{J}_n} \sum_{j\in\mathbb{M}} p_{ij}^k, \forall i \in \mathbb{M}, n \in \mathbb{N}. \tag{32}$$

Thus, Eq. (23) can be written as

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

which indicates that

$$\mu_{in}^2 + \sum_{\substack{n'\in\mathbb{N}\\ n'\neq n}} \mu_{in'}\mu_{in} x_{n'n} \le \mu_{in}\tilde{J}_n, \tag{34}$$

and thus,

$$\sum_{n=1}^{l} \mu_{in}^2 + \sum_{n=1}^{l} \sum_{\substack{n'\in\mathbb{N}\\ n'\neq n}} \mu_{in'}\mu_{in} x_{n'n} \le \left(\sum_{n=1}^{l} \mu_{in}\right) \tilde{J}_l, \tag{35}$$

where the last inequality comes from Eq. (31). Note that for any $n \le l$ and $n' > l$, we have $x_{n'n} = 0$, thus

$$\sum_{n=1}^{l} \sum_{\substack{n'\in\mathbb{N}\\ n'\neq n}} \mu_{in'}\mu_{in} x_{n'n} = \frac{1}{2}\left(\sum_{n=1}^{l} \mu_{in}\right)^2 - \frac{1}{2}\sum_{n=1}^{l} \mu_{in}^2. \tag{36}$$

Combining Eqs. (35) and (36), we have

$$\frac{1}{2}\sum_{n=1}^{l}\mu_{in}^2+\frac{1}{2}\left(\sum_{n=1}^{l}\mu_{in}\right)^2\le\left(\sum_{n=1}^{l}\mu_{in}\right)\tilde{J}_l, \tag{37}$$

which indicates that

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

Similarly, we have

$$\tilde{J}_l\ge\frac{1}{2}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}}p_{ij}^k,\forall l\in\mathbb{N},j\in\mathbb{M}. \tag{39}$$

Combining Eqs. (38) and (39), we have

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

which completes the proof. □

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

Lemma 4.3. *In MCS algorithm, the job completion time of the lth job* $J_l\le(2M+1)\tilde{J}_l$.

Proof. Let's consider an easier case that all jobs are released 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

$$\begin{aligned}J_l&\le\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\max\left\{\max_{i\in\mathbb{M}}\sum_{j\in\mathbb{M}}p_{ij}^k,\max_{j\in\mathbb{M}}\sum_{i\in\mathbb{M}}p_{ij}^k\right\}\\&\le\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}\sum_{i\in\mathbb{M}}\sum_{j\in\mathbb{M}}p_{ij}^k\le2M\tilde{J}_l.\end{aligned} \tag{41}$$

Note that the first inequality in Eq. (41) 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 4.2. When jobs have arbitrary release time, Eq. (41) can be written as

$$J_l\le r_l+2M\tilde{J}_l, \tag{42}$$

because the idle time introduced by release time can be upper-bounded by r_n. Taking Eq. (26) into consideration, we have

$$J_l\le\tilde{J}_l+2M\tilde{J}_l=(2M+1)\tilde{J}_l, \tag{43}$$

which completes the proof. □

From Lemma 4.3, the solution of our approximation algorithm can be represented as

$$\begin{aligned}SOL&=\sum_{l=1}^{N}w_lJ_l\le(2M+1)\sum_{l=1}^{N}w_l\tilde{J}_l\\&=(2M+1)OPT_{LP},\end{aligned} \tag{44}$$

and the approximation ratio

$$\alpha=\frac{SOL}{OPT_{LP}}\le2M+1, \tag{45}$$

which completes the proof. □

However, this bound is not tight, because for the last inequalities in Eqs. (40) and (41), 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.

4.3. Extensions

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

Corollary 4.1. *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.*,

$$\sum_{j\in\mathbb{M}}p_{ij}^k=\sum_{i\in\mathbb{M}}p_{ij}^k=c_k,\forall i,j\in\mathbb{M},k\in\mathbb{K}, \tag{46}$$

Eq. (40) can be transformed into

$$\tilde{J}_l^*\ge\frac{1}{2}\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}c_k, \tag{47}$$

while Eq. (43) can be transformed into

$$J_l^*\le r_l+\sum_{n=1}^{l}\sum_{k\in\mathbb{J}_n}c_k\le\tilde{J}_l^*+2\tilde{J}_l^*=3\tilde{J}_l^*, \tag{48}$$

thus

$$\alpha^*=\frac{SOL^*}{OPT_{LP}^*}=\frac{\sum_{l=1}^{N}w_lJ_l^*}{\sum_{l=1}^{N}w_l\tilde{J}_l^*}\le3, \tag{49}$$

which completes the proof. □

Corollary 4.2. *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. By setting $r_l=0$ for all $l\in\mathbb{N}$. □

5. Oversubscribed topology

5.1. Oversubscription in datacenters

In datacenters, switches are usually organized into two or three layers, for example, core, (aggregation,) and ToR, from top to below. The term *oversubscription* is defined as a property that the total bandwidth of end hosts is larger than the total bandwidth of core switches, and we use the term *fan-in factor* to describe the ratio between them [35]. Consider a simple example: there are 2 clusters connected via a 10Gbps link, each of which contains 30 hosts connected via 1Gbps links; the fan-in factor can be calculated as $\frac{60\times1\text{Gbps}}{2\times10\text{Gbps}}=3:1$, which means that when all of the intra-rack links are busy, at most $\frac{1}{3:1}\approx33\%$ of them run inter-rack traffics.

Oversubscription is quite common in datacenters, and the fan-in factor can vary from 2:1 to 20:1 for different datacenters with all kinds of traffic patterns [17,18,36,37]. The main reason for oversubscription is to save cost, and it's reported that a datacenter with 20,000 hosts can save about 7,000,000 dollars on equipments with a 3:1 oversubscribed design [35]. However, oversubscribed topologies is not easy to analysis. With non-blocking hypothesis, network congestion can only occur at end hosts; but in oversubscribed topologies, congestion can be found anywhere, including end hosts and in-network switches, which is hard to bi-model them. As a

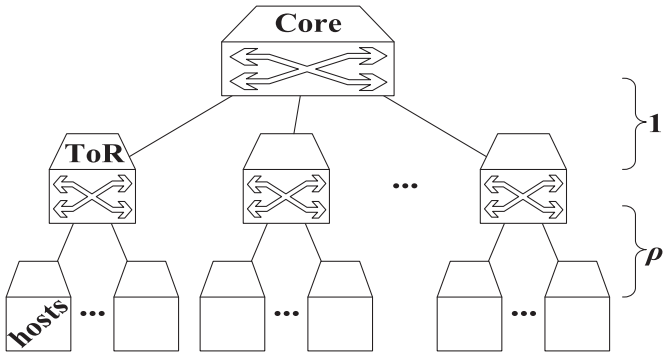


Fig. 2. The 2-layer oversubscribed networks with fan-in factor ρ.

Table 3
Notations used in adapted formulation.

Symbol	Definition
R	the number of racks
H	the number of hosts in each rack
$\mathbb{R} = \{1, 2, \cdots, R\}$	the rack set
$\gamma_i \in \mathbb{R}$	the rack that host i belongs to
ρ	the fan-in factor of an oversubscribed topology
ρ_{ij}^k	the shadow size (Definition 5.2) of flow f_{ij}^k

result, none of existing theoretical works considered this scenario.

Comparing with oversubscribed topologies in production (*e.g.*, Fig. 6 in [18]), we simplify the production topology by merging some switches as a single logical switch to avoid multi-path routing. Specifically, we regard the whole core layer as a core switch, and regard each pod as a ToR switch.

5.2. Algorithm adaption

We abstract the oversubscribed topologies as a two-layer switching network, and denote the fan-in factor as ρ: 1, as shown in Fig. 2. The notations we used in the following adapted formulation are summarized in Table 3, all of which are constant prior information, instead of decision variables. Before reformulating for oversubscribed topologies, we first introduce two definitions as follows.

Definition 5.1 (η-worst-case completion time)**.** Considering a per-flow fair-sharing transport protocol in oversubscribed networks, the η-worst-case completion time is the completion time of the coflow $\mathbf{C} = \eta \begin{bmatrix} \mathbf{0} & \mathbf{I}_{(R-1)H} \\ \mathbf{I}_H & \mathbf{0} \end{bmatrix}$ when there is no other coflow in networks, where $\mathbf{I}_n \in \mathcal{R}^{n \times n}$ is the identity matrix.

Definition 5.2 (Shadow size)**.** For a flow f_{ij}^k with size p_{ij}^k, its shadow size is the size of a virtual flow whose completion time is equal to the p_{ij}^k-worst-case completion time when host i and j do not belong to the same rack; otherwise, its shadow size is exactly its size.

To sum up, the shadow size of a flow is a normalized size when there are only inter-rack traffics sharing the core links, such that the completion time of real-size flow in this oversubscribed topology is equal to that of corresponding shadow-size flow in a non-blocking topology. We use the notion ρ_{ij}^k to denote the shadow size of f_{ij}^k. Easy to see that when host i and j do not belong to the same rack (*i.e.*, connected by two different ToR switches), the shadow size $\rho_{ij}^k = \rho p_{ij}^k$. Again, ρ is the fan-in factor of the oversubscribed topology. The adapted formulation **(A)** is as follows.

$$\textbf{(A)} \ \min \sum_{n=1}^{N} w_n J_n \tag{50}$$

s.t. (2)(3)(5)(6)(7)(8)(9);

$$\int_{r_n}^{F_{ij}^k} f_{ij}^k(t)dt = \rho_{ij}^k, \forall i, j \in \mathbb{M}, n \in \mathbb{N}, k \in \mathbb{J}_n; \tag{51}$$

$$T = \sum_{i \in \mathbb{M}} \sum_{j \in \mathbb{M}} \sum_{k \in \mathbb{K}} \rho_{ij}^k + \max_{n \in \mathbb{N}} r_n; \tag{52}$$

$$\rho_{ij}^k = p_{ij}^k, \forall i, j \in \mathbb{M}, k \in \mathbb{K}, \gamma_i = \gamma_j; \tag{53}$$

$$\rho_{ij}^k = \rho p_{ij}^k, \forall i, j \in \mathbb{M}, k \in \mathbb{K}, \gamma_i \neq \gamma_j. \tag{54}$$

Now, we can relax **(A)** to a linear programming in a similar way, as illustrated in Section 3.3. To adapt the approximation algorithm to oversubscribed topologies, we just need to replace the real size of each flow (*e.g.*, p_{ij}^k) with the shadow size (*e.g.*, ρ_{ij}^k) when calculating, then transform the shadow size to corresponding real size in each coflow-slice after IBvN decomposition and transmit them one by one. The remaining things are the same as Algorithms 1 and 2.

Corollary 5.1. *For oversubscribed networks, all of the theorems and corollaries about approximation ratios in Section 4 hold.*

Proof. By replacing all p_{ij}^k in Section 3 and Section 4 with ρ_{ij}^k. □

Discussion: The shadow size abstraction is an adaption instead of an accurate formulation. The shadow size is an abstraction of inter-rack traffics, but intra-rack traffics is not well modeled. For some workloads which contain rich intra-rack traffics, our adapted algorithm cannot make the best use of the bottleneck link. But overall, it performs well in practice.

6. Implementation

We have implemented our algorithm in an event-based flow-level simulator. Compared with real environments, protocol details is abstracted due to efficiency concerns. For example, data are treated as continuous flows instead of discrete packets, in addition, flow/congestion control and other guarantees provided by TCP are neglected. To measure the difference and check the correctness, or even take a small step to make an offline algorithm more practical, a testbed implementation is needed.

6.1. Testbed

We build two small scale testbeds to evaluate our MCS algorithm in both non-oversubscribed scenarios and oversubscribed scenarios, individually, where each testbed contains 7 hosts: one master host and 6 worker hosts. We use the master host as a global scheduler, which runs the MCS algorithm and determines the behaviors of the worker hosts. Each host is equipped with 2 eight-core Intel E5-2630 2.40GHz CPUs, 128GB memory, an Intel X710 NIC, an Intel CX-4 NIC, and a 4TB disk.

All of these hosts are connected by a management network; besides, the worker hosts are connected by another network, namely, the data network. Note that the two networks are independent with each other, thus no switching node is shared by them. The management network is organized with a big switch topology, and all 7 hosts are connected to a switch via an 1Gbps link. For non-oversubscribed scenarios, the data network is organized with the big switch topology; for oversubscribed scenarios, the data network is organized with a dumbbell topology. Mellanox SN2700

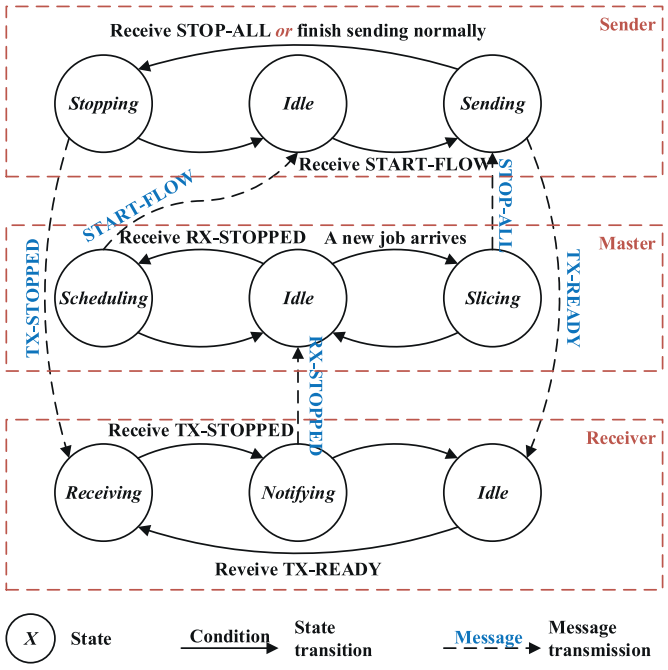


Fig. 3. The state machine of scheduling framework.

switches are used in data networks, and work in both 1Gbps and 10Gbps modes.

The operating system in each host is Ubuntu 14.04.3 LTS with the 3.19.0 Linux kernel, and uses DCTCP as the transport congestion control algorithm. Besides, we enable the RED queues for data network switches to support DCTCP, while parameters recommended in [38] are used. Unless otherwise specified, the per-port switch buffer size (*i.e.*, the maximum queuing length of each egress port in switches) is 1MB.

6.2. Scheduling framework

To evaluate our algorithms in testbed, we design and implement an app-layer scheduling framework that schedules coflows and delivers control messages. Our framework is composed by 3 components: the master component that runs in the master host, the sender component and the receiver component that run in each worker host. The state machine can be found in Fig. 3, and in the next, we will introduce the details.

Our MCS algorithm is not an online algorithm, thus we assume that the information about all coflows are known in advance by the master host; besides, before the time that any job arrives, the preprocessing works (Line 1 and Line 2 in Algorithm 1) have been down. Initially, the states of all components are *Idle*. When a new job arrives, the state of the master is transformed from *Idle* to *Slicing*, then broadcast STOP-ALL messages to all worker hosts. During the period of waiting for workers' replies, the master recalculates coflow-slices with IBvN decomposition algorithm (Line 13 in Algorithm 1) and saves them in a slice table, then transforms its state back to *Idle*. When a sender receives the STOP-ALL message, there are two possible cases:

- If the state of the sender is *Sending, i.e.*, there is an active flow that is sending data to a receiver, the state will be transformed to *Stopping* and connection will be closed by senders. Meanwhile, a TX-STOPPED message will be delivered to the receiver. After that, the sender transforms its state to *Idle*.
- If the state of the sender is *Idle, i.e.*, there is no active flows, the STOP-ALL message will be ignored.

When a receiver receives the TX-STOPPED message (or a data flow finished normally), the state of the receiver will be transformed from *Receiving* to *Notifying*. After all of the on-the-fly bytes are correctly received, the receiver will send an RX-STOPPED message to the master, and transform its state to *Idle*.

The master maintains a state table that indicates whether a sender or a receiver is idle. When receiving the RX-STOPPED message, the master will update the item of corresponding sender and receiver in state table. When finding an item with both idle sender and idle receiver in slice table, the master will transform its state from *Idle* to *Scheduling*. After sending a FLOW-START message to the corresponding sender and updating the tables, the master will transform its state back to *Idle*. For a sender that receives the FLOW-START message, the state of the sender will be transformed from *Idle* to *Sending*, and a TX-READY message will be sent to the corresponding receiver. When the receiver receives the TX-READY message, it will transform its state from *Idle* to *Receiving*, then a TCP connection between them will be established to transmit data.

Except for the external mathematical tools, our scheduling framework is implemented in Java. We use the Socket package provided in Java for data flow generating, and use Akka [39] to deliver the control messages. Note again that, the data flows are transmitted in the data network, and the control messages are transmitted in the management network. Akka is an implementation of the famous actor model in JVM, which is used to handle the parallel processing in datacenters.

7. Evaluation

We evaluate our algorithm with both a small-scale testbed and a large-scale 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 [7,9,12,16,40].

7.1. Methodology

Settings: For there is no prior work aimed at multi-stage coflow scheduling to minimize 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 guarantee [9]. 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 [15,16]. We enable work conservation for all algorithms to guarantee a fair comparison.

Workload: The coflow information in Facebook logs is incomplete. For each coflow, the Facebook logs contain its sender hosts, receiver hosts, 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 jitter 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. To evaluate our algorithms in the scenarios of non-trivial dependencies, for a job with c coflows, at most $c-1$ dependencies are generated randomly and independently to form a DAG, which may contain more than one connected component. 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 than 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 83.58%, 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 hosts (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.

7.2. Testbed results

To investigate the performance of our MCS algorithm in a realistic environment, we use two small scale testbeds (non-oversubscribed, oversubscribed) for evaluation in both 1Gbps and 10Gbps modes, denoted as MCS(1G) and MCS(10G) in Fig. 4, respectively. We use a subset of Facebook logs that contains 40 coflows as testbed workloads to match the testbed scale, and the coflow size are scaled by 10 × in 10Gbps mode to make the results comparable to those in 1Gbps mode. We compare our algorithm (*i.e.*, DCTCP with MCS) with pure DCTCP (*i.e.*, DCTCP without MCS), denoted as DCTCP(1G) and DCTCP(10G); besides, the corresponding simulation results and lower bounds will also be given to verify the correctness of our simulators, denoted as MCS(S) and MCS(LB), respectively. Note that we do not implement Aalo and LP-OV-LS in our testbeds, instead, we leave these comparisons in simulations.

We first evaluate our algorithms in non-oversubscribed topologies, and the results of the average JCT and the total weighted JCT are shown in Fig. 4(a) and (b), respectively. Compared with pure DCTCP, the average or total weighted JCT is reduced by up to 81.65% by our algorithms; besides, the performance gap in two modes is quite small (less than 4%), which implies that our scheduling framework is robust to bandwidth. The performance in oversubscribed topologies is shown in Fig. 4(c), which has a similar tendency with that in non-oversubscribed topologies. It is noteworthy that the performance of DCTCP(10G) is much worse than DCTCP(1G), which is because larger workloads lead to more retransmissions and longer backoff time when there are too much concurrent flows; however, our scheduling framework guarantees the congestion-free property in each host, thus the number of concurrent flows in bottleneck link is limited. We use Fig. 4(d) to evaluate this property on a 10Gbps oversubscribed topology with different per-port switch buffer size. As the per-port buffer size growing, the performance of DCTCP(10G) becomes better due to less packet-drops, while the performance of MCS(10G) keeps stable, because there is nearly no packet-drop all the time in our scheduling framework.

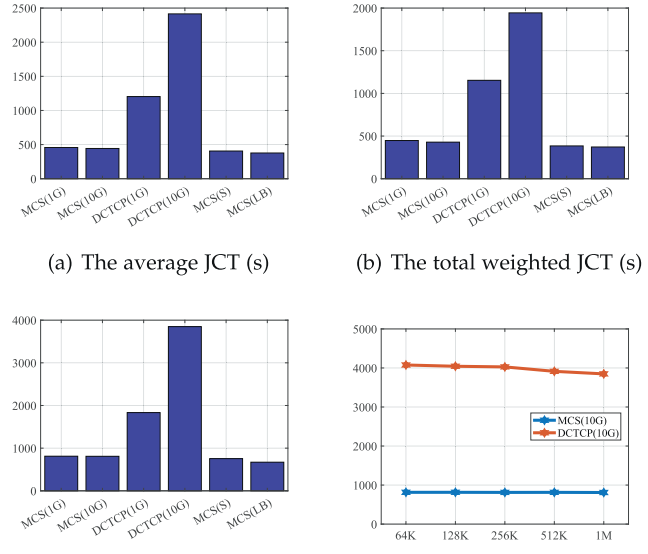
Note that the results in testbeds are slightly larger than those in simulations, which is caused by the additional delays (*e.g.*, message transmission, TCP handshaking) and bandwidth wastes (*e.g.*, TCP slow start, ACK packets).

7.3. Simulation overview

We evaluate our MCS algorithm by comparing with Aalo and LP-OV-LS in non-oversubscribed topologies (*i.e.* the big switch topology). Results are shown in Fig. 5.

The objective of total weighted completion time is not supported by Aalo. Thus, 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. Unless otherwise specified, we choose $M = 30, \theta = 30, \alpha = 20$ as the default parameters when comparing with Aalo.

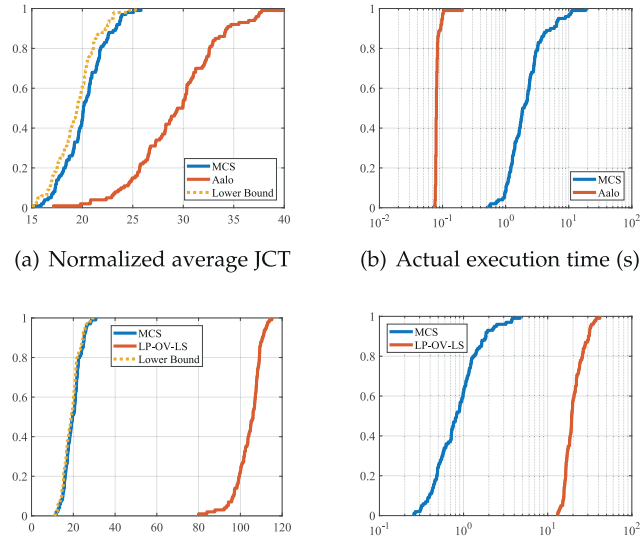
As illustrated, all of the jobs are randomly combined by the coflows in Facebook logs with random dependencies, while their release time is randomly generated, thus it's necessary to investigate the cumulative distribution function (CDF) of the average JCT and the time spent on scheduling. Shown in Fig. 5(a), our MCS algorithm performs better than Aalo, and the performance is more stable: for our algorithm, the average JCT varies from about 15 to 26, with coefficient of variation $CV_{MCS} = 0.1077$; for



(a) The average JCT (s) (b) The total weighted JCT (s)

(c) The oversubscribed topology (d) The influence of switch buffer

Fig. 4. Testbed results.



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

(c) Normalized total weighted JCT (d) Actual execution time (s)

Fig. 5. CDF of JCT and actual execution time.

Table 4
Details of coflow completion time.

	Average/weight CCT	Reverse pairs
MCS(average)	38.93(6.43)	–
Aalo	40.15(9.03)	21.15%(5.24%)
MCS(weighted)	11.71(2.99)	–
LP-OV-LS	52.93(6.77)	46.05%(3.51%)

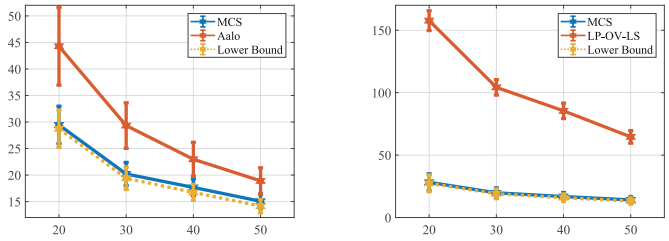
Aalo, it varies from about 17 to 40, and the coefficient of variation $CV_{Aalo} = 0.1462$. However, our algorithm is slower than Aalo. As a heuristic algorithm, Aalo schedules coflows with simple rules rather than complex computation, thus its execution time concentrates on about 0.08s; by contrast, 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 2s on average.

Next, 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. Unless otherwise specified, we choose $M = 30, \theta = 30, \alpha = 6$ as the default parameters.

Let's investigate the CDF of the total weighted JCT and the time spent on scheduling. As shown in Fig. 5(c), the total weighted JCT of LP-OV-LS is over $5\times$ 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 $20\times$ slower than our algorithm, because it has to solve a linear programming with more variables and more constraints, as shown in Fig. 5(d).

Discussion: One may wonder the reason why our MCS algorithm performs so closely to its lower bound, which is much better than theoretical results. On the one hand, the factor M in approximation ratio is introduced when we analysis the lower bound of **(ILP)** (namely, LB_1), but **(ILP)** is already a lower bound of original formulation **(O)** (namely, LB_2). The lower bound LB_2 is quite hard to analysis, so we use an easier bound LB_1 in our theoretical works. However, LB_1 is underestimated and lacks of practical meanings, thus in evaluation, we use LB_2 as a much tighter bound, which is computable in practice. On the other hand, MCS benefits from backfilling (*i.e.*, work-conserving), which is a common mechanism used in datacenters to keep the network busy. With the help of backfilling, the network throughput cannot be hurt severely by scheduling. Again, the backfilling mechanism is enabled for all evaluated algorithms to guarantee a fair comparison.

One may also wonder the reason why MCS algorithm can lead to lower JCT. We investigate the completion time of coflows, and summary the details in Table 4: the average/weighted CCT is normalized with corresponding CCT of MCS, while the reverse pair indicates the percentage of coflow pairs whose completion time is reversed compared with MCS. The CCT of MCS is similar to that of Aalo, while more than 20% coflow pairs are reversed, which means that the order of coflows plays an important role in minimizing JCT. However, the CCT of LP-OV-LS is much larger (even though it is specifically designed to minimize CCT), and the percentage of inverse pairs is high (due to different objectives). We have analyzed simulation logs and tried to find out the reason why its CCT tends to the worst case. In LP-OV-LS, a flow is allowed to transmit only if both of its sender and receiver are all idle; meanwhile, when the scheduler finds an idle sender-receiver pair, among flows with the same sender and receiver, the flow with the highest priority will be chosen to transmit. Consider a prioritized flow, only when its sender and receiver become idle (nearly) at the same time could it be chosen (it can hardly happen); otherwise, the one that becomes idle firstly is very likely to match another host, thus it will be occupied by another flow immediately and becomes no longer idle. As a result, the prioritized flow can wait for a really long time in most cases, and efforts are brought to naught.



(a) The average JCT (b) The total weighted JCT

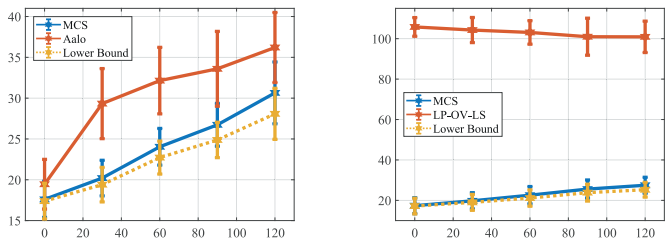
Fig. 6. The influence of cluster scale.

7.4. All kinds of scenarios

The number of hosts: First, we investigate the influence of the number of hosts, and results are shown in Fig. 6(a) and (b). Note that the total bandwidth grows linearly as the number of hosts growing, thus for all of the algorithms, the average JCT decreases. It's clear that the average JCT is not inversely proportional to the number of hosts, due to the non-uniformity of coflow size, time distribution and space distribution. Comparing with Aalo, we reduce the average JCT by 33.48%, and the largest gap between our algorithm and its lower bound is only 6.44%. Comparing with LP-OV-LS, we reduce the total weighted JCT by 82.11%, and the largest gap between our algorithm and its lower bound is only 6.63%.

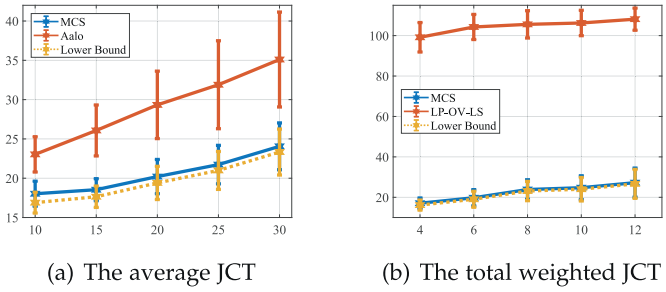
Network loads: Then we investigate the influence of the average interval of network loads, *i.e.* the parameter θ in Poisson process, shown in Fig. 7(a). A smaller θ corresponds to a heavier load. When all coflows arrived at the same time, *i.e.*, $\theta = 0$, the link is always fully used for a long 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. One may wonder the reason why the JCT decreases as the load growing. In fact, this is caused by the definition of JCT used in formulations, shown in Eqs. (2), (3) and (4). Comparing with Aalo, we reduce the average JCT by 31.10%, and the largest gap between our algorithm and its lower bound is 9.14%.

However, Fig. 7(b) is confused that the total weighted JCT of LP-OV-LS should decrease 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. Comparing with LP-OV-LS, we reduce the total weighted JCT by



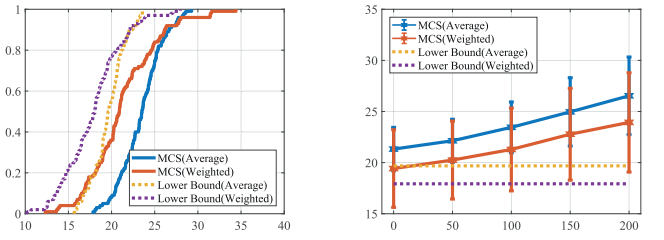
(a) The average JCT (b) The total weighted JCT

Fig. 7. The influence of network loads.



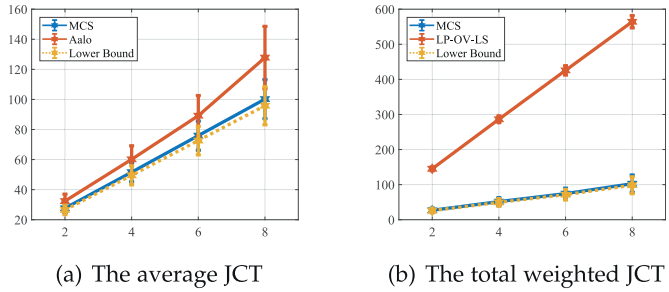
(a) The average JCT (b) The total weighted JCT

Fig. 8. The influence of job structure.



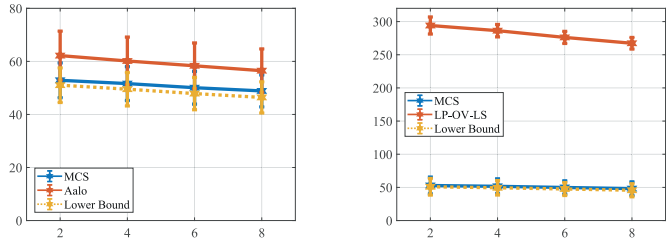
(a) CDF of JCT (b) Influence of comp. delay (s)

Fig. 9. Influence of average coflow computation delay.



(a) The average JCT (b) The total weighted JCT

Fig. 10. The influence of fan-in factors.



(a) The average JCT (b) The total weighted JCT

Fig. 11. The influence of racks.

83.58%, and the largest gap between our algorithm and its lower bound is 8.79%.

The complexity of job structure: Next, let's investigate the influence of the job structure, *i.e.*, the average number of coflows in each job. From Fig. 8(a) and (b), we can see that all of the 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. Comparing with Aalo, we reduce the average JCT by 31.87%, and the largest gap between our algorithm and its lower bound is 5.07%. Comparing with LP-OV-LS, we reduce the total weighted JCT by 82.78%, and the largest gap between our algorithm and its lower bound is 5.75%.

Computation delay: We further evaluate our algorithm in the scenarios when each coflow has a computation phase, as shown in Fig. 9. 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. 9(a). Then we evaluate our algorithm within a large range of computation delay from 0s to 200 s, where the time cost on computation is comparable with that cost on network transmission. Results are shown in Fig. 9(b): as the time cost on computation growing, the average and total weighted JCT increases as expected. By comparing with Fig. 5(a) and (c), we can see that even when 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.

7.5. Oversubscribed topologies

In oversubscribed scenarios, we evaluate our adapted algorithms by comparing with Aalo and LP-OV-LS. Noting that both Aalo and LP-OV-LS suppose that there is no in-network congestion thus do not support the oversubscribed topologies, we adapt the two algorithms using the shadow size concept in a similar way. Unless otherwise specified, we assume there are 4 racks in the whole network, and each rack contains 12 hosts, thus there are 48 hosts in total; besides, we choose the default fan-in factor as 4. Other settings are the same as non-oversubscribed scenarios.

We first investigate the influence of fan-in factors, and results are shown in Fig. 10. As expected, the JCT of all three algorithms grows linearly as the fan-in factor increasing. A larger fan-in factor means a lower core-network bandwidth, which harms the inter-rack communications thus leads to a worse application performance. To sum up, comparing with Aalo, we reduce the average JCT by 21.58%, and the largest gap between our algorithm and its lower bound is 4.59%; comparing with LP-OV-LS, we reduce the average JCT by 82.52%, and the largest gap between our algorithm and its lower bound is 4.57%.

Then we evaluate the influence of the number of racks. The total number of hosts is kept as a constant, thus the number of hosts in each rack varies across the number of racks. Shown in Fig. 11, the JCT decreases slightly as the number of racks growing; we believe it is caused by the inaccuracy of the shadow size abstraction, as is discussed in Section 5. But overall, the influence of the number of racks is negligible. To sum up, comparing with Aalo, we reduce the average JCT by 14.96%, and the largest gap between our algorithm and its lower bound is 5.27%; comparing with LP-OV-LS, we reduce the average JCT by 82.03%, and the largest gap between our algorithm and its lower bound is 5.45%.

Note that in oversubscribed scenarios, both the performance improvements and gaps are quite similar with the results in non-oversubscribed scenarios, which matches our theoretical conclusions.

8. 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 and implement it as a scheduling framework. Evaluation results show that our algorithm significantly outperforms state-of-the-art works in both testbeds and simulations.

Conflicts of interest

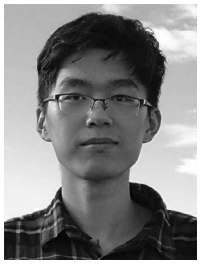
None.

Acknowledgments

This work was supported in part by the National Key R&D Program of China 2018YFB1003202, the National Natural Science Foundation of China under 61672276, 61602194, 61772265, and 61802172, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

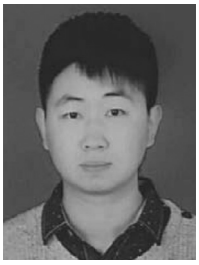
[1] B. Tian, C. Tian, H. Dai, B. Wang, Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time, in: IEEE INFOCOM 2018-IEEE Conference on Computer Communications, IEEE, 2018, pp. 864–872.
[2] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[3] Apache hadoop., 2019. (http://hadoop.apache.org).
[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, 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. 2–2.
[5] M. Chowdhury, M. Zaharia, J. Ma, M.I. Jordan, I. Stoica, Managing data transfers in computer clusters with orchestra, in: ACM SIGCOMM, vol. 41, ACM, 2011, pp. 98–109.
[6] M. Chowdhury, I. Stoica, Coflow: a networking abstraction for cluster applications, in: ACM Hotnets, ACM, 2012, pp. 31–36.
[7] M. Chowdhury, Y. Zhong, I. Stoica, Efficient coflow scheduling with varys, in: ACM SIGCOMM, ACM, 2014, pp. 443–454.
[8] F.R. Dogar, T. Karagiannis, H. Ballani, A. Rowstron, Decentralized task-aware scheduling for data center networks, in: ACM SIGCOMM Computer Communication Review, vol. 44, ACM, 2014, pp. 431–442.
[9] M. Chowdhury, I. Stoica, Efficient coflow scheduling without prior knowledge, in: ACM SIGCOMM Computer Communication Review, vol. 45, ACM, 2015, pp. 393–406.
[10] Tpc-ds., 2019. (http://www.tpc.org/tpcds).
[11] H. Susanto, H. Jin, 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.
[12] Z. Qiu, C. Stein, 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.
[13] S. Khuller, 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.
[14] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, L. Li, Towards practical and near-optimal coflow scheduling for data center networks, IEEE Trans. Parallel Distrib. Syst. 27 (11) (2016) 3366–3380.
[15] M. Shafiee, J. Ghaderi, An improved bound for minimizing the total weighted completion time of coflows in datacenters, arXiv:1704.08357 (2017a).
[16] M. Shafiee, J. Ghaderi, Brief announcement: a new improved bound for coflow scheduling, in: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2017.
[17] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, S. Sengupta, Vl2: a scalable and flexible data center network, in: ACM SIGCOMM 2009, vol. 39, ACM, 2009, pp. 51–62.
[18] 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.
[19] T.A. Roemer, A note on the complexity of the concurrent open shop problem, J. Scheduling 9 (4) (2006) 389–396.
[20] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.R. Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Ann. Discrete Math. 5 (1979) 287–326.
[21] J.Y.-T. Leung, H. Li, M. Pinedo, Order scheduling in an environment with dedicated resources in parallel, J. Scheduling 8 (5) (2005) 355–386.
[22] A. Agnetis, H. Kellerer, G. Nicosia, A. Pacifici, Parallel dedicated machines scheduling with chain precedence constraints, Eur. J. Oper. Res. 221 (2) (2012) 296–305.
[23] M. Queyranne, Structure of a simple scheduling polyhedron, Math. Program. 58 (1–3) (1993) 263–285.
[24] A.S. Schulz, Scheduling to minimize total weighted completion time: performance guarantees of lp-based heuristics and lower bounds, in: International Conference on Integer Programming and Combinatorial Optimization, Springer, 1996, pp. 301–315.
[25] L.A. Hall, D.B. Shmoys, J. Wein, Scheduling to minimize average completion time: Off-line and on-line algorithms, in: SODA, vol. 96, 1996, pp. 142–151.
[26] Y.-A. Kim, Data migration to minimize the total completion time, J. Algorithms 55 (1) (2005) 42–57.
[27] R. Gandhi, M.M. Halldórsson, G. Kortsarz, H. Shachnai, Improved bounds for scheduling conflicting jobs with minsum criteria, ACM Trans. Algorithms (TALG) 4 (1) (2008) 11.
[28] W. Rodiger, T. Muhlbauer, P. Unterbrunner, A. Reiser, A. Kemper, T. Neumann, Locality-sensitive operators for parallel main-memory database clusters, in: Data Engineering (ICDE), 2014 IEEE 30th International Conference on, IEEE, 2014, pp. 592–603.
[29] C.-S. Chang, D.-S. Lee, C.-Y. Yue, Providing guaranteed rate services in the load balanced birkhoff-von neumann switches, IEEE/ACM Trans. Netw. (TON) 14 (3) (2006) 644–656.
[30] J.K. Sundararajan, S. Deb, M. Médard, Extending the birkhoff-von neumann switching strategy for multicast-on the use of optical splitting in switches, IEEE J. Sel. Areas Commun. 25 (6) (2007) 36–50.
[31] M. Marcus, R. Ree, Diagonals of doubly stochastic matrices, Q. J. Math. 10 (1) (1959) 296–302.
[32] F. Dufossé, B. Uçar, Notes on birkhoff–von neumann decomposition of doubly stochastic matrices, Linear Algebra Appl. 497 (2016) 108–115.
[33] L.G. Khachiyan, A polynomial algorithm in linear programming, USSR Comput. Math. Math. Phys. 20 (80) (1979) 1–3.
[34] 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.
[35] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, in: ACM SIGCOMM Computer Communication Review, vol. 38, ACM, 2008, pp. 63–74.
[36] Cisco data center infrastructure 2.5 design guide., 2019. (https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5a_book/DCInfra_2a.html).
[37] N. Farrington, A. Andreyev, Facebook data center network architecture, IEEE Optical Interconnects Conf, Citeseer, 2013.
[38] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data center tcp (dctcp), in: in Proc. ACM SIGCOMM 2011, 41, ACM, 2011, pp. 63–74.
[39] Akka., 2019, (http://akka.io).
[40] Y. Li, S.H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, 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.



Bingchuan Tian received the B.S. degree in the department of computer science and technology from Nanjing University of Aeronautics and Astronautics, China, in 2016. He is working towards the Ph.D. degree in the department of computer science and technology in Nanjing University, China. His research interests include datacenter networking and networked systems.

Chen Tian is an associate professor at State Key Laboratory for Novel Software Technology, Nanjing University, China. He was previously an associate professor at School of Electronics Information and Communications, Huazhong University of Science and Technology, China. Dr.Tian received the BS (2000), MS (2003) and PhD (2008) degrees at Department of Electronics and Information Engineering from Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming and urban computing.

Bingquan Wang received the B.S. degrees from the Department of Computer Science and Technology at the Southeast University, China, in 2016. He is a 3rd-year M.S. student in Nanjing University, China. His research interests include distributed networks and network architecture. He is a student member of the IEEE. Bo Li received the B.S. degree from the department of Computer Science and Engineering at the Nanjing University of Science and Technology, China, in 2016. He is a 3rd-year M.S. student in Nanjing University, China. His research interests include distributed networks and systems.



Bo Li received the B.S. degree from the department of Computer Science and Engineering at the Nanjing University of Science and Technology, China, in 2016. He is a 3rd-year M.S. student in Nanjing University, China. His research interests include distributed networks and systems.

Zehao He received the B.S. degree from the Department of Computer Science and Technology at the Shandong University, China, in 2018. He is working toward the M.S. degree in the Department of Computer Science and Technology at Nanjing University. His research interests include datacenter networks and systems.

Haipeng Dai received the B.S. degree in the department of electronic engineering from Shanghai Jiao Tong University, Shanghai, China, in 2010, and the Ph.D. degree in the department of computer science and technology in Nanjing University, Nanjing, China, in 2014. He is a research assistant professor in the department of computer science and technology in Nanjing University. He is an IEEE and ACM member. He received Best Paper Award from IEEE ICNP5, Best Paper Award Candidate from IEEE INFOCOM'17.

Kexin Liu received the B.S. degree from the Department of Software Engineering at Sun Yatsen University, China, in 2017. She is working toward the M.S. degree in the Department of Computer Science and Technology at Nanjing University, China. Her research interests include datacenter networks and network architecture. Wanchun Dou received the Ph.D. degree in mechanical and electronic engineering from the Nanjing University of Science and Technology, China, in 2001. He is currently a Full Professor of the State Key Laboratory for Novel Software Technology, Nanjing University. From April 2005 to June 2005 and from November 2008 to February 2009, he respectively visited the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, as a Visiting Scholar. Up to now, he has chaired three National Natural Science Foundation of China projects and published more than 60 research papers in international journals and international conferences. His research interests include workflow, cloud computing, and service Computing.

Wanchun Dou received the Ph.D. degree in mechanical and electronic engineering from the Nanjing University of Science and Technology, China, in 2001. He is currently a Full Professor of the State Key Laboratory for Novel Software Technology, Nanjing University. From April 2005 to June 2005 and from November 2008 to February 2009, he respectively visited the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, as a Visiting Scholar. Up to now, he has chaired three National Natural Science Foundation of China projects and published more than 60 research papers in international journals and international conferences. His research interests include workflow, cloud computing, and service computing.

Guihai Chen is a distinguished professor of Nanjing University. He earned B.S. degree in computer software from Nanjing University in 1984, M.E. degree in computer applications from Southeast University in 1987, and Ph.D. degree in computer science from the University of Hong Kong in 1997. He had been invited as a visiting professor by Kyushu Institute of Technology in Japan, University of Queensland in Australia and Wayne State University in USA. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture and data engineering. He has published more than 350 peer-reviewed papers, and more than 200 of them are in well-archived international journals such as IEEE TPDS, IEEE TC, IEEE TKDE, ACM/IEEE TON and ACM TOSN, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNext and AAAI. He has won 9 paper awards including ICNP 2015 best paper award and DASFAA 2017 best paper award.