Selective Coflow Completion for Time-sensitive Distributed Applications with Poco

Shouxi Luo
Southwest Jiaotong University

Pingzhi Fan
Southwest Jiaotong University

Huanlai Xing
Southwest Jiaotong University

Hongfang Yu
University of Electronic Science and Technology of China

ABSTRACT

Recently, the abstraction of *coflow* is introduced to capture the collective data transmission patterns among modern distributed data-parallel application. During processing, coflows generally act as barriers; accordingly, time-sensitive applications prefer their coflows to complete within deadlines and deadline-aware coflow scheduling becomes very crucial.

Regarding these data-parallel applications, we notice that many of them, including *large-scale query system*, *distributed iterative training*, and *erasure codes enabled storage*, are able to tolerate loss-bounded incomplete inputs by design. This tolerance indeed brings a flexible design space for the schedule of their coflows: when getting overloaded, the network can trade coflow completeness for timeliness, and balance the completenesses of different coflows on demand. Unfortunately, existing coflow schedulers neglect this tolerance, resulting in inflexible and inefficient bandwidth allocations.

In this paper, we explore this fundamental trade-off and design Poco, a POlicy-based COflow scheduler, to achieve customizable selective coflow completions for these emerging time-sensitive distributed applications. Internally, Poco employs a suite of novel designs along with admission controls to make *flexible*, *work-conserving*, and *performance-guaranteed* rate allocation to online coflow requests very efficiently. Extensive trace-based simulations indicate that Poco is highly flexible and achieves optimal coflow schedules respecting the requirements specified by applications.

CCS CONCEPTS

• **Networks** → **Data center networks**; **Traffic engineering algorithms**.

KEYWORDS

coflow, deadline, flow scheduling, policy

ACM Reference Format:
Shouxi Luo, Pingzhi Fan, Huanlai Xing, and Hongfang Yu. 2020. Selective Coflow Completion for Time-sensitive Distributed Applications with Poco. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3404397.3404449

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404449

1 INTRODUCTION

In modern cloud data centers, distributed data-parallel applications such as Hadoop, Spark, and EC-Cache, are widely employed to build large-scale data processing, analysis, and storage services [4, 16]. In these systems, a job is split into multiple staged tasks carried out by a cluster in distributed manners. During processing, involved servers trigger groups of parallel, collective flows to move intermediate results from machines of the current stage to the next. These flows in the same group are abstracted as a *coflow* since they share the same performance goal and their completions act as the barrier of the distributed computation [4, 5]. For time-sensitive applications like web search, retail, recommendation systems, etc., the triggered coflows are generally bound with deadlines, implying the dates by which they should be finished [5, 12]. To deal with these transfers, existing deadline-aware (co)flow scheduling proposals directly reject a request if its deadline can not be met [5, 17]; or admit all requests then dynamically preempt large-sized, less-emergency transfers in service to increase the amount of deadline-satisfied requests heuristically, without performance guarantee [11–13]. Unfortunately, such designs are proven to be sub-optimal for many of emerging distributed applications.

Due to the approximate nature of the involved distributed computation [7, 21], or the redundant design employed for data transmission [10, 16], many of today's distributed applications are able to tolerate incomplete transmissions by design. For instance, in *large-scale query systems* like web search and advertisement selection, for each cache-missed request, a group of backend servers will report then aggregate their top-N results to generate the final response; a partial data transmission is acceptable to the application since it is a sample for the whole data thus still bringing benefits to the application [7]. Likewise, during the *distributed iterative training* of modern machine learning models, besides the tolerance of incomplete training data, models like *deep neural network* based image classification and natural language understanding, are robust to achieve comparable convergence rate over incomplete parameter updates [21]—Actually, the recent empirical study of [20] shows that many machine learning algorithms are bounded-loss tolerant; their end-to-end job performance would get little impact in case the randomized network data loss is below a certain fraction (typically 10%∼35%). And for applications like *erasure codes enabled distributed storage system*, on object reads/downloads, because of the redundant self-coding designs, obtaining any k out of $(k + r)$

splits of the object residing in the cluster, are sufficient to serve the request [10, 16].

All the above observations demonstrate the ubiquity of tolerance on incomplete inputs among emerging distributed applications. Recently, by using this type of tolerance, Liu *et al.* propose a protocol with controlled packet loss called ATP, to perform approximate data transmission for approximate application [9]; Xia *et al.* design BTP, a Bounded-loss Tolerant transport Protocol, to remove the tail latency for the parameter synchronization process of distributed model training [20]. However, both ATP and BTP are oblivious of the coflow semantic among flows; their per-flow based designs are proved to be sub-optimal for the schedule of coflow [5]. To support incompleteness-tolerant coflow scheduling, Im *et al.* employ greedy designs to maximize the partial throughput of coflow [7]. However, the proposed Con-Myopic algorithm is unaware of the application requirements in terms of the exact (coflow) completeness and timeliness. As a result, Con-Myopic provides no performance guarantee to time-sensitive application. Moreover, Con-Myopic assumes that the data transmitted by a flow cannot be replaced by another. This is not always true as the aforementioned applications show counter-examples. For those applications, the data transmitted by all or portions of the flows in a coflow is exchangeable. Accordingly, the completeness of these flows is described by the total volume they deliver successfully. In these cases, the schedule of Con-Myopic is inflexible and inefficient.

In summary, emerging time-sensitive distributed data-parallel applications are common to tolerate incomplete yet loss-bounded inputs. This brings an import yet overlooked design space for the schedule of their deadline-bounded coflows: in case the network is overloaded thus impossible to complete all tasks in time, we could trade coflow completeness for timeliness and trade one coflow's completeness for those of others.

This work. In this paper, we explore the fundamental trade-off between the time a coflow could take to complete and the completeness it would achieve. As different applications generally have various requirements on the completeness and timeliness of coflow, we extend the barrier definition of coflow to support partial completion and develop Poco, a POlicy-based COflow scheduler, to achieve customizable selective completion for them. To provide guaranteed performances, Poco involves admission controls for coflows arriving online. At the high-level, it provides a set of policy primitives, with which, distributed applications can precisely define their requirements of both the expired time and minimum completeness along with each coflow. Then, at the low-level, Poco translates these requirements into time-slotted linear constraints and formulate a Linear Program (LP) to solve. If the corresponding LP is infeasible, Poco rejects the request; otherwise, any feasible result of the problem yields a bandwidth allocation to admit the new request without sacrificing the requirements of others.

However, building LPs for the selective-completion schedule of coflow and solving them for rate scheduling are quite challenging. Firstly, coflow requests arrive online; although a coflow's detailed requirements would be available upon its arrival but it is hard to get that information ahead of time [14, 19]; thus, greedily allocating all available bandwidth to admit an incoming request would be unfair to future requests, resulting in unfairness among their applications. Secondly, as we will show, the bandwidth allocation suggested by LP might be non-work-conserving; Poco should not directly use the raw LP results for rate controls. Thirdly, as an online scheduler, the solving of involved LPs must be very efficient.

To address these challenges, Poco *i*) employs a tunable model to control the level at which bandwidth in the future is allocated in admission control; *ii*) designs a post processing to make work-conserving bandwidth allocations; *iii*) merges variables to compact the model, and more essentially, *iv*) develops a parallelizable core to speed up the LP solving by making use of the specific constraint structures of the problem. Extensive evaluations confirm its flexibility and performance gains.

Limits of Poco. As a centralized scheduler, Poco introduces scheduling delays. In practice, a coflow's actual duration depends on both the available network bandwidth and the amount of data it should deliver. For those small coflows that could complete within a very short time (e.g., one or two RTTs), the scheduling delay of Poco might be not negligible thus Poco could not help. In practice, there also exist many distributed applications like BSP-based distributed machine learning and user-facing approximate bigdata analytics whose triggered coflows are bulk and such delays are acceptable [5, 12, 18]. Poco is mainly designed for them.

Contributions. To sum up, we make these contributions:

- An analysis of the design space and desired proprieties of deadline-aware, loss-bounded coflow schedulers (§2).
- A high-level coflow abstraction along with a LP model that enables applications to express completeness requirements for deadline-sensitive coflows (§3.1, §3.2).
- A suite of schedule designs to compress the model size and make fair yet work-conserving bandwidth allocations for coflows incoming online (§3.3).
- An efficient solver accelerating the LP solving by leveraging the specific structure of its constraints (§4).
- Extensive trace-based evaluations assess the feasibility, effectiveness, and scalability of Poco (§5).

2 POCO GUIDELINES

To start the design, let us first analyze the design space raised by the tolerance of emerging distributed application (§2.1), and summarize the desirable properties of Poco (§2.2).

2.1 Design Space

Consider that a group of flows $\mathcal{F}_e$ go through the same bottleneck link e, and the sending rate of flow f at time t is $r_f(t)$. Obviously, as (1) shows, the volume that f can deliver before time t is determined by the integration of its allocated sending rate $r_f(t)$ over time, which is restrained by the rates allocated to all other flows in turn as (2) indicates. Motivated by this, we obtain a foundational design space for the schedule of coflow: in a heavily-loaded network, by taking advantage of the application's tolerance of incomplete inputs, we can *i*) trade the achieved completeness for shorter completion times, and *ii*) trade one flow's completeness for those of others. Moreover, if the data delivered by a group of flows within a coflow ($\mathcal{F}_g$ for instance) is exchangeable, the network can balance

the total task (ϕ_g for instance) among its sub-flows with respect to the network loads as (3) indicates.

$$v_f = \int_0^t r_f(t)dt \tag{1}$$

$$r_f(t) \le c_e - \sum_{f' \in \mathcal{F}_e \setminus \{f\}} r_{f'}(t) \tag{2}$$

$$\sum_{f \in \mathcal{F}_g} v_f \ge \phi_g \tag{3}$$

2.2 Desirable Properties

By exploring aforementioned trade-offs, Poco performs selective coflow completions for time-sensitive applications. To be practical, it must realize the following design goals.

Performance guarantees. First of all, to ensure the progress of distributed computation, applications usually have limited level of tolerance. Hence, Poco should provide a service model with performance promises to applications.

High flexibility. Second, different applications are likely to have various performance requirements. Accordingly, Poco needs to be flexible enough to support various requirements.

Fairness. Third, coflows arrive online; the requirements of future coflow requests are agnostic ahead of time. Greedily allocating all available bandwidth to admit requests is unfair to future arrivals [8]. Thus, Poco should support configurable admission control.

Work-conserving. Fourth, to fully utilize the network and serve more requests, Poco is required to be work-conserving. That is to say a link sits idle only if there is no traffic demand.

Scalability. Last but not least, as an online scheduler, Poco must decide whether to admit a request and schedule all flow sending rates to guarantee their performances efficiently. For this purpose, the algorithms employed by Poco must run in real-time with low time complexity.

3 POCO SCHEDULER

As Figure 1 sketches, Poco employs admission controls to provide promises of completenesses and deadlines for coflows in the online scenario. On getting an incoming request, if Poco finds a way to meet its completeness- and deadline- requirements without violating those of any existing coflow, this new request could be admitted and a corresponding bandwidth allocation is already found. Otherwise, the request would get rejected; the application could either cancel the request, or relax its requirements then resubmit again.

In this section, we first introduce the coflow abstraction (§3.1) along with the network model (§3.2) Poco provides to capture the flexible requirements raised by application, then describe the optimization designs that Poco adopts to achieve *fairness*, *work-conserving*, and *scalability* (§3.3).

3.1 Coflow Abstraction

As Figure 2 summarizes, Poco abstracts a coflow request, saying C_i for instance, by the set of its involved flows $\mathcal{F}_i = \{f_{i,1}, f_{i,2}, \cdots\}$, and the group of its associated completeness requirements $\mathcal{R}_i = \{\cdots, (G_{i,k}; \phi_{i,k}), \cdots\}$. Compared with the original coflow abstraction proposed by [4, 5], Poco mainly extends the coflow model to support partial completion. For the j-th subflow in $\mathcal{F}_i$, i.e., $f_{i,j}$, its task is to transmit data with remaining volume $v_{i,j}$ via established path $p_{i,j}$ within expired time $\tau_{i,j}$. Although our model allows $\tau_{i,j}$ vary among flows, in practice, a coflow represents a task and thus flows belonging to the same coflow generally share the same deadline τ_i. In case the network is overloaded and a very strict hard deadline is desired, it is impossible to make full transmissions of all flows within their deadlines. Then, Poco makes selectively loss-bounded partial completions. The k-th restriction in $\mathcal{R}_i$ given by the application, specifies that the total completed volume of flows in $G_{i,k}$ should not be less than $\phi_{i,k}$.

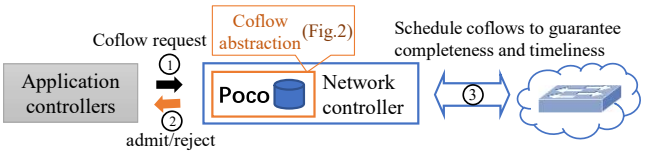


Figure 1: Service model of Poco

Grammar

C_i	::=	$(\mathcal{F}_i; \mathcal{R}_i)$	Application-specified coflow request
$\mathcal{F}_i$	::=	$\{\cdots, f_{i,j}, \cdots\}$	Transfer demands of cofow C_i
$\mathcal{R}_i$	::=	$\{\cdots, (G_{i,k}; \phi_{i,k}), \cdots\}$	Completeness requirements
$f_{i,j}$	::=	$(\tau_{i,j}; v_{i,j}; p_{i,j})$	Details of the j-th subflow in coflow i

More Notation

$\tau_{i,j}$	:	Expired time of flow $f_{i,j}$ (we have $\forall j : \tau_{i,j} = \tau_i$ in this paper)
$v_{i,j}$	:	Remaining volume of flow $f_{i,j}$
$p_{i,j}$	:	Path of flow $f_{i,j}$
$G_{i,k}$	:	Set of flow(s) in the same completeness group
$\phi_{i,k}$	:	Completeness requirement

Figure 2: The coflow abstraction provided by Poco.

Obviously, the abstraction provided by Poco is very expressive. With it, applications can specify coflow requests along with both timeliness- and completeness- requirements easily. For transfers without deadlines, Poco simply treats them bound with a very loose expired time. And for coflows unable to tolerate incompleteness, Poco sets their exact volumes as the completeness requirements.

3.2 Network Model

Without loss of generality, consider that there are $n-1$ accepted yet uncompleted coflow requests, labeled $C_1, \cdots, C_{n-1}$, and the newly incoming request to check is C_n. We assume that bandwidth is allocated in time slots with length Δ_T and we denote the rate of flow $f_{i,j}$ during time slot t by $r_{i,j,t}$. Then, the problem of finding a bandwidth allocation to admit request $\mathcal{R}_n$ and meet its requirements without violating those of others, is straightforward to be formulated as the system of linear inequalities shown in (4). Here, $v_{i,j}$ and $\phi_{i,k}$ are the updated remaining flow size and uncompleted completeness volume requirement, respectively; $c_{e,t}$ denotes the available capacity of link e at time slot t that can be allocated to admit the currently incoming request now.

$$(4)\begin{cases} \sum_{(i,j)\in G_{i,k}} \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T \geq \phi_{i,k}, \quad \forall i,k & (4a) \\ \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T \leq v_{i,j}, \quad \forall i,j & (4b) \\ \sum_{(i,j):e\in p_{i,j}} r_{i,j,t} \leq c_{e,t}, \quad \forall e,t & (4c) \\ r_{i,j,t} \geq 0, \quad \forall i,j,t & (4d) \end{cases}$$

It is obvious that, if constraints in are infeasible, the request must be rejected; otherwise, any feasible $\{r_{i,j,t}\}$ satisfying (4) yields a bandwidth allocation that accepts C_n.

3.3 Schedule Designs

On getting a request, the straightforward design of Poco is to *i*) formulate the associated bandwidth allocation problem as a Linear Program (LP) by introducing trivial objectives such as maximizing the total completed volume subject to the constraints of (4), as (5) show;[1] then *ii*) employ commercial off-the-shelf optimizer to solve, and *iii*) finally perform the admission control and rate schedule based on the results. However, such a design is impractical, since *i*) the bandwidth of time slot in future might be over-allocated, resulting in unfairness to future arrivals; *ii*) more seriously, the rate scheduling given by the LP does not guarantee work-conserving; and last but not least, *iii*) the LP model involves too many variables, making the process of model solving time costly.

$$\text{Maximize} \sum_{i=1}^{n} \sum_{j=1}^{|F_i|} \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T \ s.t.\ (4) \tag{5}$$

Fairness. To be fair for future coflow arrivals, Poco systematically limits the allocation of link capacities in future time slots on performing admission control. Suppose that link e is with the capacity of c_e, motivated by the design of [8], Poco lets $c_{e,t} = c_e\beta(t)$, in which $\beta(t) = \min(1, exp(-(t-t_*)/t_o))$, t_* and t_o are two tunable parameters, receptively. By tuning them, Poco can control the level at which future link capacities are allocated. Note that, to be work-conserving in practice, for admitted requests, Poco should allocate all link capacities to serve until they complete or expire.

Work-conserving. As we will show, the rate scheduling suggested by LPs does not guarantee work-conserving, even if fine-grained timeslots are employed and a very large t_* is used in $\beta(t)$. For instance, consider that an active flow would expire at time 1 and two coflows C_i and C_j will appear at time 0 and 1, then expire at the same time 2, respectively. Accordingly, let T_Δ be one unit of time; then there are two time slots, t_1 with range $[0,1)$ and t_2 with range $[1,2)$. Suppose that each of these two incoming coflows involves only one subflow, saying $f_{i,1}$ and $f_{j,1}$, going through the same bottleneck link with capacity 2. The total volume and completeness requirement of $f_{i,1}$ are 2 and 1, respectively, while those of $f_{j,1}$ are 3 and 2, respectively. At time 0, the corresponding LP for the admission control of coflow C_i is as (7) shows. By solving the problem with either simplex or interior-point method, we might get the result of $r_{i,1,1} = 0, r_{i,1,2} = 2$ (indeed, this is exactly the solution given by Mosek 8.1.67 [1], a commercial off-the-shelf LP solver), yielding a bandwidth allocation to admit coflow C_i. However, such a scheduling is not work-conserving since no traffic occurs in slot t_1. As a result, at time 1, coflow C_j would get rejected since there does not exist enough bandwidth to guarantee its requirements. For this specific instance, it is possible to achieve work-conserving bandwidth allocation by assigning degressive weights to slotted rates in the objective (e.g., $\sum_{i=1}^{n}\sum_{j=1}^{|F_i|}\sum_{t=1}^{\tau_{i,j}} \frac{r_{i,j,t}}{\tau_{i,j}}$). However, such a design is impractical as Poco must under-allocate bandwidth in future slots on admission control for fairness.

$$(6)\begin{cases} 0 \leq r_{i,1,1} + r_{i,1,2} \leq 2 & (6a) \\ 0 \leq r_{i,1,1} \leq 2 & (6b) \\ 0 \leq r_{i,1,2} \leq 2 & (6c) \end{cases}$$

$$\text{Maximize } r_{i,1,1} + r_{i,1,2} \ \ s.t.\ (6) \tag{7}$$

To address this, Poco employs a post process to adjust the rate schedules given by LP. Basically, if there is remaining bandwidth in earlier slots, Poco greedily moves parts of a slot's task up, until no movement can be made. At the end, a work-conserving rate scheduling is obtained. In case there are multiple flows going through the same under-loaded link, flows with unmet completeness requirements would occupy the available bandwidth before those whose completeness requirements are already satisfied; and for either requirement- unmet or met flows, residual slotted link capacities are allocated to them in non-decreasing order of their deadlines.

Scalability. Because of the fine-grained slotted bandwidth allocation, the model involves a large number of variables, taking non-trivial time for LP solvers to deal with. In addition, the aforementioned process of work-conserving might also introduce significant delays since the number of slots to be checked could be huge. To overcome these, *i*) Poco lets flows that already meet their completeness requirements have the rate of 0 for model pruning, and merges successive time slots between flow expiration events into a single one. Then, *ii*) Poco employs a novel algorithm to solve the compacted LP in parallel by leveraging the specific structure of its constraints. Finally, *iii*) Poco modifies the results given by LP solver to make work-conserving adjustments. Next, we describe how Poco merges time slots and performs post progresses and leave the detail of its parallel solver to §4.

On performing admission controls, if no flow expires from time slot t_1 to t_2, it is reasonable to assume that all flow rates keep consistent during the interval, without impacting either the feasibility or optimality of the problem. Such a design is equivalent to adding the constraints of $r_{i,j,t} = r_{i,j,t-1}$ for unexpired flows $\{f_{i,j} : \forall \tau_{i,j} > t\}$ and slots $\{t : \nexists \tau_{i,j} = t\}$. Following this, merged time slots are with various lengths and the number of variables for the schedule of flows with N diverse deadlines would be limited to $N(N+1)/2$, which is independent from either flow lifespans or the setting of slot width. Denote the l-th expired time as $\tau^{(l)}$ and let $\tau^{(0)}$ be 0.

[1] In this paper, Poco employs the objective of maximizing the total completed volume as a case study. With standard reformulation techniques [2], it is very easy to extend Poco to support other types of schedules like maximizing the minimal gain of achieved completeness in a max-min fashion: i.e., Maximize $\min_{\forall i,k} \frac{1}{\phi_{i,k}} \sum_{(i,j)\in G_{i,k}} \sum_{t=1}^{\tau_{i,j}} r_{i,j,t}\Delta_T$.

Then, for the l-th merged time slot, starting from $\tau^{(l-1)}$ to $\tau^{(l)}$, the corresponding available capacity of link e for admission control, is corrected as $\beta_l c_e$, in which β_l is the corresponding correct factor of link capacity defined in (10). Accordingly, the problem of (5), along with its constraints (4), can be reformulated to (9), where $\pi_{i,j}$ is the interval index of the diverse expired time $\tau_{i,j}$ (11), $\Delta_T^{(l)}$ is the width of the l-th merged slot (12), and $\bar{r}_{i,j,l}$ is the rate of $f_{i,j}$ during that interval. Here, N is the number of diverse deadlines, which is also the amount of time slotted after merging.

$$(8)\begin{cases} \sum\limits_{(i,j)\in G_{i,k}} \sum\limits_{l=1}^{\pi_{i,j}} \Delta_T^{(l)} \bar{r}_{i,j,l} \geq \phi_{i,k}, \quad \forall i,k & \text{(8a)} \\ \sum\limits_{l=1}^{\pi_{i,j}} \Delta_T^{(l)} \bar{r}_{i,j,l} \leq v_{i,j}, \quad \forall i,j & \text{(8b)} \\ \sum\limits_{(i,j):e\in p_{i,j}} \bar{r}_{i,j,l} \leq \beta_l c_e, \quad \forall e,l & \text{(8c)} \\ \bar{r}_{i,j,l} \geq 0, \quad \forall i,j,l & \text{(8d)} \end{cases}$$

$$\text{Maximize} \sum_{i=1}^{n} \sum_{j=1}^{|F_i|} \sum_{l=1}^{\pi_{i,j}} \bar{r}_{i,j,l} \Delta_T^{(l)} \ \ s.t.\ (8) \tag{9}$$

$$\beta_l \ := \ \frac{\sum_{t=\tau^{(l-1)}}^{\tau^{(l)}-1} \beta(t)}{\tau^{(l)} - \tau^{(l-1)} + 1} \tag{10}$$

$$\pi_{i,j} \ := \ \underset{l}{\text{argmax}}\{l : \tau^{(l)} \leq \tau_{i,j}\} \tag{11}$$

$$\Delta_T^{(l)} \ := \ (\tau^{(l)} - \tau^{(l-1)})\Delta_T, \ l = 1, \cdots, N \tag{12}$$

As for the post bandwidth adjustment, Poco repeats to fully fill available slotted link capacities by moving parts of a slot's task up until no link capacity is left. Since the result of LP yields a bandwidth allocation to the future, Poco could work in pipeline by keeping making work-conserving adjustments for slots only in the near future to keep low process delay.

4 EFFICIENT POCO SOLVER

To support large-scale selective coflow scheduling, Poco employs a parallelizable solver to solve (9) by making use of the specific *sparse* and *block-angular* constraint structure of the involved LP model. Basically, the solver of Poco is built upon the well-known *homogeneous model based primal-dual interior-point method* (HPD-IPM) [1, 2]. Next, we first overview the workflow of HPD-IPM on solving our problem (§4.1), then describe the detail of how Poco reduces and parallelizes the involved matrix computations (§4.2).

4.1 Workflow of HPD-IPM

Given (9), we can rewrite it in its matrix format as (13) shows, where T stands for the operator of transpose. The exact formats of $\boldsymbol{A}$, $\boldsymbol{x}$, and $\boldsymbol{b}$ of (8) follow in §4.2.

$$\text{Minimize } \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} \ \ s.t.\ \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{13}$$

1 Choose starting point $\boldsymbol{z}^0 : (\boldsymbol{x}^0; \boldsymbol{y}^0; \boldsymbol{s}^0; \theta^0; \kappa^0)$, parameter ε_f, ε_g, γ, and η
2 **for** $k \leftarrow 0$ **to** $maxIter - 1$ **do**
3 $\quad \boldsymbol{r}_b^k \leftarrow \boldsymbol{b}\theta^k - \boldsymbol{A}\boldsymbol{x}^k$
4 $\quad \boldsymbol{r}_c^k \leftarrow \boldsymbol{w}\theta^k - \boldsymbol{A}^T\boldsymbol{y}^k - \boldsymbol{s}^k$
5 $\quad \boldsymbol{r}_d^k \leftarrow \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}^k + \kappa^k - \boldsymbol{b}^T\boldsymbol{y}^k$
6 $\quad \mu^k \leftarrow \frac{\left(\boldsymbol{x}^k\right)^T \boldsymbol{s}^k + \theta^k\kappa^k}{rank(\boldsymbol{A})+1}$
7 $\quad$ **if** $\mu^k \leq \varepsilon_g$ ***and*** $\left\|\left(\boldsymbol{r}_b^k; \boldsymbol{r}_c^k; \boldsymbol{r}_d^k\right)\right\| \leq \varepsilon_f$ **then**
8 $\quad\quad$ **break**
9 $\quad$ Compute $\boldsymbol{d}$: $(\boldsymbol{d_x}; \boldsymbol{d_y}; \boldsymbol{d_s}; d_\theta; d_\kappa)$ by solving[a]

$$\begin{bmatrix} \boldsymbol{A} & & & -\boldsymbol{b} & \\ & \boldsymbol{A}^{\mathsf{T}} & I & -\boldsymbol{w} & \\ -\boldsymbol{w}^{\mathsf{T}} & \boldsymbol{b}^{\mathsf{T}} & & & -1 \\ \boldsymbol{S}^k & & \boldsymbol{X}^k & & \\ & & & \kappa^k & \theta^k \end{bmatrix} \begin{bmatrix} \boldsymbol{d_x} \\ \boldsymbol{d_y} \\ \boldsymbol{d_s} \\ d_\theta \\ d_\kappa \end{bmatrix} = \begin{bmatrix} \eta\boldsymbol{r}_b^k \\ \eta\boldsymbol{r}_c^k \\ \eta\boldsymbol{r}_d^k \\ -\boldsymbol{X}^k\boldsymbol{s}^k + \gamma\mu^k\boldsymbol{e} \\ -\theta^k\kappa^k + \gamma\mu^k \end{bmatrix}$$

$\quad$ in which $\boldsymbol{X}^k \leftarrow diag(\boldsymbol{x}^k), \boldsymbol{S}^k \leftarrow diag(\boldsymbol{s}^k)$
10 $\quad$ Calculate and choose the step size α^k
11 $\quad \boldsymbol{z}^{k+1} \leftarrow \boldsymbol{z}^k + \alpha^k\boldsymbol{d}$

Algorithm 1: The workflow of how the homogeneous algorithm [1] solves the problem of (14).

[a] Despite different variations of the algorithm might vary in detail, all they share the same core of solving equations in the form of (15).

According to the theory of linear optimization [1, 2], the problem of (13) can be solved by solving its homogeneous self-dual derivation (14), a linear program with strictly complementary solutions that can be obtained by the well-known homogeneous algorithm [1]. Then, if its strictly complementary solution $(\boldsymbol{x}^*; \boldsymbol{y}^*; \boldsymbol{s}^*; \theta^*; \kappa^*)$ has $\theta^* > 0$, $\frac{\boldsymbol{x}^*}{\theta^*}$ yields an optimal solution to (13); otherwise, the original problem is infeasible.

$$\begin{aligned} \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\theta = \ & \mathbf{o} \\ \boldsymbol{A}^{\mathsf{T}}\boldsymbol{y} + \boldsymbol{s} - \boldsymbol{w}\theta = \ & \mathbf{o} \\ \boldsymbol{b}^{\mathsf{T}}\boldsymbol{y} - \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} - \kappa = \ & \mathbf{o} \\ \boldsymbol{y} \text{ is free}, \ (\boldsymbol{x}; \boldsymbol{s}; \theta; \kappa) \geq \ & \mathbf{o} \end{aligned} \tag{14}$$

In short, HPD-IPM is a variation of the well-known primal-dual interior-point method [2]. Given the problem of (14), it starts from an initial point like $(\boldsymbol{e}; \mathbf{o}; \boldsymbol{e}; 1; 1)$, then inters for a decreasing sequencing of μ, until the result is converged or the iteration exceeds a threshold, as Algorithm 1 sketches. Obviously, the most time-costly procedure in each iteration is to solve a linear equation system (Line 9) and the workflow [1] is to first i) compute $\boldsymbol{d}_y$ from (15) for an established right hand side vector $\boldsymbol{v}$, then ii) obtain the entire $\boldsymbol{d}$, where $\boldsymbol{D}^k = \boldsymbol{X}^k(\boldsymbol{S}^k)^{-1} = diag(\frac{x_1^k}{s_1^k}, \frac{x_2^k}{s_2^k}, \cdots)$.

$$\boldsymbol{A}\boldsymbol{D}^k\boldsymbol{A}^{\mathsf{T}}\boldsymbol{d}_y = \boldsymbol{v} \tag{15}$$

Obviously, $\boldsymbol{A}\boldsymbol{D}^k\boldsymbol{A}^{\mathsf{T}}$ is a positive-semidefinite matrix that has the Cholesky decomposition of $\boldsymbol{L}\boldsymbol{L}^{\mathsf{T}}$ in most cases; we can obtain $\boldsymbol{d}_y$ by solving $\boldsymbol{L}\boldsymbol{g} = \boldsymbol{v}$ and $\boldsymbol{L}^{\mathsf{T}}\boldsymbol{d}_y = \boldsymbol{g}$, sequentially. In the following, we show how Poco achieves this in an efficient way.

4.2 Parallel Cholesky Solver

Indeed, as we will show, the $\boldsymbol{A}$ involved in Poco has the multiple-level primal block-angular structure, which makes the process of Cholesky decomposition and solving parallelizable [6]. This enables us to speed up Poco solver greatly by making use of the abundant cores in modern server.

To identify the block-angular structure of $\boldsymbol{A}$, let us revisit the constraints specified in (8). Notably, the corresponding requirements to admit a new coflow request are made of two classes: *i*) meeting each coflow's minimum transfer demands within their deadlines (i.e., (8a)) without excusing the maximum size of each flow (i.e., (8b)); and *ii*) making sure that no link gets overloaded through the transmitting (i.e., (8c)). Let Γ be the set of all time-slotted link resources that would be used by coflow, and further denote the corresponding link and time index of the o-th time-slotted link resource κ_o as κ_o^e and κ_o^t, respectively. Then, the associated time-slotted link capacity Poco could use for admission control now can be computed by $\beta_{\kappa_o^t} c_{\kappa_o^e}$. By using $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$ to indicate these two types of constraint matrices for coflow i, respectively, the linear constraints shown in (8) can be reformed as a primal block-angular structure as (16) shows, in which $\boldsymbol{x}^s$ is the vector of slack variables for the constraints of link capacity at each time interval, and the details of $\boldsymbol{x}_i$, $\boldsymbol{b}_i$, and $\boldsymbol{b}^*$ follow in (17), (18), and (19).

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} \boldsymbol{A}_1 & & & & \\ & \boldsymbol{A}_2 & & & \\ & & \ddots & & \\ & & & \boldsymbol{A}_n & \\ \boldsymbol{B}_1 & \boldsymbol{B}_2 & \cdots & \boldsymbol{B}_n & I \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \\ \boldsymbol{x}^s \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \vdots \\ \boldsymbol{b}_n \\ \boldsymbol{b}^* \end{bmatrix} \tag{16}$$

$$\begin{aligned} \boldsymbol{x}_i \coloneqq [&(\bar{r}_{i,1,1}, \cdots, \bar{r}_{i,1,\pi_{i,1}}, \hat{r}^+_{i,1}), \\ &\cdots, \\ &(\bar{r}_{i,|F_i|,1}, \cdots, \bar{r}_{i,|F_i|,\pi_{i,|F_i|}}, \hat{r}^+_{i,|F_i|}), \\ &\hat{r}^-_{i,1}, \cdots, \hat{r}^-_{i,|\mathcal{R}_i|}]^\mathsf{T} \end{aligned} \tag{17}$$

$$\boldsymbol{b}_i \coloneqq \left[v_{i,1}, \cdots, v_{i,|F_i|}, \phi_{i,1}, \cdots, \phi_{i,|R_i|}\right]^\mathsf{T} \tag{18}$$

$$\boldsymbol{b}^* \coloneqq \left[\beta_{\kappa_1^t} c_{\kappa_1^e}, \cdots, \beta_{\kappa_{|\Gamma|}^t} c_{\kappa_{|\Gamma|}^e}\right]^\mathsf{T} \tag{19}$$

As (17) shows, $\boldsymbol{x}_i$ is made up of the rate allocations of flows belonging to coflow i, along with a few slack variables $\{\hat{r}^+_{i,j}\}$ and $\{\hat{r}^-_{i,j}\}$. Likewise, as (20) indicates, the constraint matrix involved by each coflow also follows exactly the same pattern of block-angular structure, in which, $\psi_{i,k,j}$ is either 1 or 0, indicating whether the k-th requirement $G_{i,k}$ involves $f_{i,j}$ or not. As for $\boldsymbol{B}_i$, constant $\hbar_{o,i,j,l}$ is either 1 or 0, indicating whether $f_{i,j}$ goes though link κ_o^e during time interval κ_o^t (24). Here, we also have $\kappa_o^t \equiv l$.

$$\boldsymbol{A}_i \coloneqq \left[\begin{array}{cccc|ccc|c} \boldsymbol{a}_{i,1} & 1 & & & & & & \\ & \boldsymbol{a}_{i,2} & 1 & & & & & \\ & & & \ddots & & & & \\ & & & & \boldsymbol{a}_{i,|F_i|} & 1 & & \\ \hline \psi_{i,1,1}\boldsymbol{a}_{i,1} & 0 & \cdots & & \psi_{i,1,|\mathcal{F}_i|}\boldsymbol{a}_{i,1,|F_i|} & 0 & -1 & \\ \vdots & \vdots & \vdots & & \vdots & \vdots & & \ddots \\ \psi_{i,|R_i|,1}\boldsymbol{a}_{i,1} & 0 & \cdots & & \psi_{i,|R_i|,|\mathcal{F}_i|}\boldsymbol{a}_{i,|\mathcal{F}_i|} & 0 & & -1 \end{array}\right] \tag{20}$$

$$\boldsymbol{a}_{i,j} \coloneqq [\Delta_T^{(1)}, \cdots, \Delta_T^{(\pi_{i,j})}] \tag{21}$$

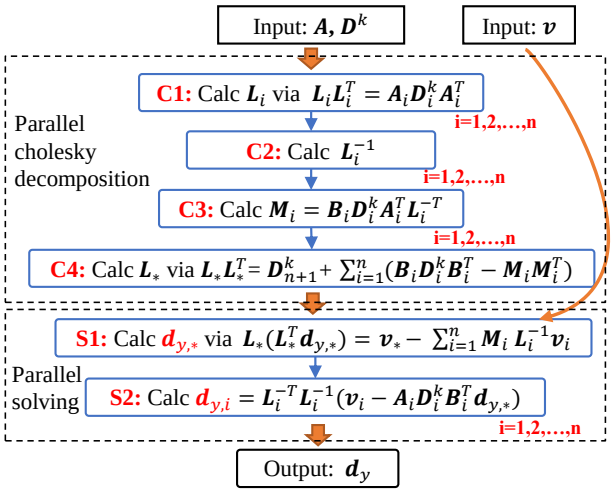


Figure 3: Workflow of how Poco solves (15) in parallel

$$\psi_{i,k,j} \coloneqq \begin{cases} 1 & (i,j) \in G_{i,k} \\ 0 & otherwise \end{cases} \tag{22}$$

$$\boldsymbol{B}_i \coloneqq \left[\begin{array}{ccc|c|ccc|c} \cdots & \hbar_{1,i,j,1} & \cdots & \hbar_{1,i,j,\pi_{i,j}} & 0 & \cdots & & \\ \ddots & \vdots & \ddots & \vdots & \vdots & \ddots & & \boldsymbol{o} \\ \cdots & \hbar_{|\Gamma|,i,j,1} & \cdots & \hbar_{|\Gamma|,i,j,\pi_{i,j}} & 0 & \cdots & & \end{array}\right]_{j=1,\cdots,|F_i|} \tag{23}$$

$$\hbar_{o,i,j,l} \coloneqq \begin{cases} 1 & \kappa_o^e \in p_{i,j} \wedge l \le \pi_{i,j} \\ 0 & otherwise \end{cases} \tag{24}$$

Recall that the core of HPD-IPM is to solve $\boldsymbol{A}\boldsymbol{D}^k\boldsymbol{A}^\mathsf{T} = \boldsymbol{v}$ via Cholesky decomposition techniques. By splitting the diagonal $\boldsymbol{D}^k$ into blocks and let the i-th block $\boldsymbol{D}_i^k$ shares the same shape with $\boldsymbol{A}_i$, we obtain that $\boldsymbol{A}\boldsymbol{D}^k\boldsymbol{A}^T$ has the form

$$\boldsymbol{A}\boldsymbol{D}^k\boldsymbol{A}^T = \begin{bmatrix} \boldsymbol{A}_1\boldsymbol{D}_1^k\boldsymbol{A}_1^T & & & \boldsymbol{A}_1\boldsymbol{D}_1^k\boldsymbol{B}_1^T \\ & \boldsymbol{A}_2\boldsymbol{D}_2^k\boldsymbol{A}_2^T & & \boldsymbol{A}_2\boldsymbol{D}_2^k\boldsymbol{B}_2^T \\ & & \ddots & \vdots \\ & & \boldsymbol{A}_n\boldsymbol{D}_n^k\boldsymbol{A}_n^T & \boldsymbol{A}_n\boldsymbol{D}_n^k\boldsymbol{B}_n^T \\ \boldsymbol{B}_1\boldsymbol{D}_1^k\boldsymbol{A}_1^T & \boldsymbol{B}_2\boldsymbol{D}_2^k\boldsymbol{A}_2^T & \cdots & \boldsymbol{B}_n\boldsymbol{D}_n^k\boldsymbol{A}_n^T \quad \boldsymbol{C} \end{bmatrix} \tag{25}$$

where

$$\boldsymbol{C} = \sum_{i=1}^{n} \boldsymbol{B}_i\boldsymbol{D}_i^k\boldsymbol{B}_i^\mathsf{T} + \boldsymbol{D}_{n+1}^k \tag{26}$$

It is easy to verify that Cholesky factorization always preserves this bordered form [6]. Accordingly, to be consistent with the structure of A, we label its blocks as (27) shows, and partition both the variable vector $\boldsymbol{d}_y$ and the right hand side value vector $\boldsymbol{v}$ into blocks, $\boldsymbol{d}_y \coloneqq (\boldsymbol{d}_{y,1}; \boldsymbol{d}_{y,2}; \cdots; \boldsymbol{d}_{y,n}; \boldsymbol{d}_{y,*})$, $\boldsymbol{v} \coloneqq (\boldsymbol{v}_1; \boldsymbol{v}_2; \cdots; \boldsymbol{v}_n; \boldsymbol{v}_*)$.

$$\boldsymbol{L} = \begin{bmatrix} \boldsymbol{L}_1 & & & & \\ & \boldsymbol{L}_2 & & & \\ & & \ddots & & \\ & & & \boldsymbol{L}_n & \\ \boldsymbol{M}_1 & \boldsymbol{M}_2 & \cdots & \boldsymbol{M}_n & L_* \end{bmatrix} \tag{27}$$

Then, the workflow of how Poco solves $\boldsymbol{d}_y$ in parallel is as Figure 3 shows. Notably, at the high-level, the computations involved in C1, C2, C3 and S2 can be parallelized for each coflow i; and at

the low-level, since each $\boldsymbol{A}_i$ repeats the same primal block-angle structure, the computation involved in S1 could be accelerated with the similar design. Moreover, all matrix computations are able to be parallelized as well.

Such a design has two advantages: on one hand, it reduces the computation greatly by exploiting the sparsity of $\boldsymbol{A}$; on the other, it makes the computation parallelizable at multiple levels by making use of the well structure of $\boldsymbol{A}$.

5 EVALUATION

In this section, we evaluate Poco through trace-based simulations. We compare it with state-of-the-art deadline-aware coflow schedulers Varys [5], Con-Myopic [7], and the default baseline Fair-Sharing (FS). Extensive results indicate that Poco is flexible and robust to make very efficient tolerance-aware coflow scheduling:

(1) Poco lets more coflows meet their requirements by trading the achieved completeness for timeliness, and trading one coflow's completeness for those of others;
(2) its scheduling algorithm makes very effective use of the network to provide guaranteed performance to time-sensitive coflow, outperforming that of the state-of-the-art Varys up to 1.25× and even more (the performance gain depends on the instance's settings);
(3) its core solver is very efficient, achieving linear speedup (hundreds and even more) by making usage of the specific block-angle structure of its constraint matrix.

5.1 Methodology

Workload. The coflow workloads employed in evaluations are generated using a coflow workload generator following the design provided by Varys [5]. In short, it unsamples the Facebook traces to the desired number of coflows, network load, cluster scale, etc., while keeping workload characteristics similar to the original Facebook trace. However, the Facebook trace does not involve the attribute requirements of deadline and completeness. In common with prior work [5, 12], for each coflow C_i, we set its deadline constraint to be $(1+z)\rho_i$, where ρ_i is the minimum completion time of coflow i in an empty network, and z is a randomly number following the uniform distribution $U[0; 2x]$. As for the completeness requirements, we assume that each involves the requirements: $\mathcal{R}_i : \{(\mathcal{F}_i, \alpha_i \sum_{j=1}^{|\mathcal{F}_i|} v_{i,j})\}$, where α_i varies from 0 to 1. Unless mentioned otherwise, our simulation results use the baseline of 300 coflows, 1.0 network load, 0.9 completeness requirements; and x, the scale factor of deadline (i.e., the mean of z), is set as 1.

Cluster. We find that simulations imply consistent results under diverse cluster scales. To reduce the simulation time, we consider a cluster involves 60 servers here. In common with recent work, the entire cluster network is abstracted out as a non-blocking switch [5, 15], which interconnects all machines with 1 Gbps bidirectional links.

Simulator. Similar to that of Varys, we develop flow-level simulators (in Python3) to perform detailed replays of the aforementioned coflow traces, according to the schedule policy of FS, Varys, Con-Myopic, and the proposed Poco, respectively. In short, FS is the max-min fair sharing policy adopted by TCP and its variations. Varys is the state-of-the-art deadline-guaranteed coflow scheduler, which performs admission controls by letting coflows finish exactly at their deadlines, then adjusting sending rates to achieve work-conserving [5]. Con-Myopic is the only existing scheduler designed to support partial completions; it greedy schedules coflows to maximize their marginal partial throughput without considering their exact deadlines [7]. Poco admits coflow requests based on the results of (9), then adjusts flow rates to achieve work-conserving. As for the back-end solver of Poco, we implement a parallel core based on Scipy to denote the performance gain, in which the parallelization is implemented and controlled by Numpy[2] implicitly. In all tests, rejected requests do not get resubmitted. Theoretically, a fine-grained slotted model would ensure a more efficient use the network. However, setting Δ_T too small would cause the underlying transport protocol (e.g., DCTCP) to behave erratically due to significant variation of available bandwidth. Also, smaller slots would increase the running time of simulation greatly. We suggest the use of $O(100)$ ms slots and let Δ_T be 500 ms in our simulation. As for the tunable parameter t_* and t_o for the control of available link capacity in future, we let t_* be $\tau_*/(u\Delta_T)$, and t_o be 1000, respectively, where τ_* is the 85-percentile of the involved coflows' ideal completion times and u is the average network load, both of which can be inferred from served coflows in practice.

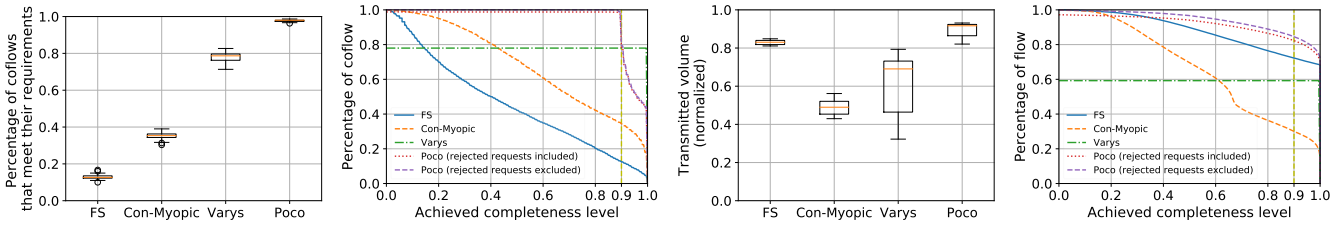
Metrics. Regarding the performance metrics, we mainly consider the percentage of coflows that meet their requirements of deadline and completeness. For specific test cases, we also consider the (normalized) completed volume and achieved completeness under various scheduling schemes. Besides, we also evaluate the speedup of Poco's core solver shown in Figure 3. For each parameter setting, we perform 8 trials.

5.2 Performance

Detailed case study. As Figure 4a shows, under the default parameter settings, FS, Con-Myopic, and Varys let about 13.1%, 35.2%, and 78.0% coflows meet their completeness and deadline requirements, respectively. In contrast, the average percentages achieved by Poco is 97.7%, yielding the performance gain of 7.46×, 2.78×, and 1.25×, respectively. For these test cases, Figure 4b gives their detailed Complementary Cumulative Distribution Function (CCDF) curves of all coflow requests. Recall that both Poco and Varys employ admission controls to provide performance guarantees; accordingly, their curves involve line segments. However, Varys neglects the tolerance nature of application and always makes full completion for admitted coflow, resulting in performance loss compared with Poco. Regarding FS and Con-Myopic, they work poorly since the agnosticism of application requirements. Meanwhile, we also observe that the schedule of Con-Myopic does not guarantee work-conserving, since it is designed to maximize the marginal partial throughput at each slot [7].

To ascertain their performance details, we also count the total transmitted volume in each case (normalized by the total volume of all requests, Figure 4c) and the achieved completeness of each flow (Figure 4d). Obviously, Poco makes very efficient use of the

[2] SciPy: Open source scientific tools for Python https://www.scipy.org



(a) Requirement-satisified coflow **(b) Achieved completeness** **(c) Transmitted volume** **(d) Achieved completeness of flow**

Figure 4: The details of Fair-Sharing (FS), Con-Myopic, Varys, and Poco on scheduling coflows with 0.9-completeness and "$x = 1$"-deadline requirements.

network as it transmits nearly 89.4% of all the volume, accounting about 92.6% of the total volume of the coflow requests it admits. As for Varys, it makes 100% deliveries for all the coflows it admits, accounting about 62.8% of the total requested volume. A very interesting observation is that, FS only lets about 13.1% requests meet their requirements, however, its transmitted volume reaches 83.0% of the total. Such results imply that, maximizing the network goodput/throughput does not necessarily optimize the completion of coflow. Thus, to perform efficient coflow scheduling, the awareness of both the completeness and deadline is a must for the scheduler. As an example, Poco enables more coflow requirements be met by trading completeness for timeliness and trading one coflow's completeness for those of others on demand. The illustration shown in Figure 4d confirms the awareness and flexibility of Poco: all admitted coflows do satisfy the completeness requirements of 0.9, yet at the flow-level, less than 86% of the admitted flows achieve the completeness level of 0.9 for their own tasks.

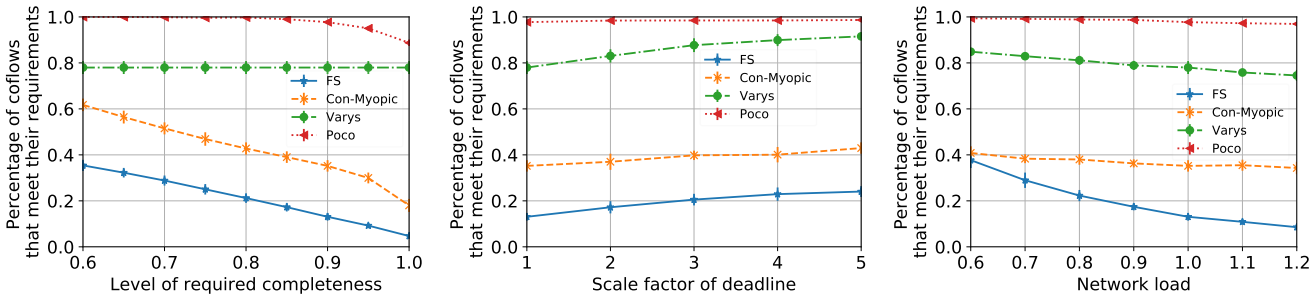
Impact of completeness. To investigate the impact of completeness, we change each α_i, the required completeness level, from 1 to 0.6, then rerun the tests and check the percentage of coflows that could meet their requirements under various schedule schemes. As Figure 5a illustrates, the results of Varys keep consistent, because it is unaware of the tolerance of completeness thus always performing 100% completions for all admitted requests. Conversely, with the relaxation of required completeness, all other three schemes schedule more coflows to meet their requirements. Especially, Poco is able to admit and satisfy all the requests, once their required completeness level is less than 0.8. Such results imply the ability of Poco on performing tolerance-aware scheduling, again. Moreover, we find that Poco still outperforms Varys about 5%, even when all coflows require 100% completions. That is to say, the rate schedule algorithm adopted by Poco always makes more efficient use of the bandwidth than that of Varys. This is reasonable, since the schedule of Poco is built upon LP and Poco obtains the optimal results in polynomial time. Besides, the results of FS and Con-Myopic also reveal that their percentages of met coflow increase linearly with the decrease of the required completeness level. This phenomenon is consistent with the observed distribution shown in Figure 4b.

Impact of deadline. As the other requirement dimension of a coflow request, we then test how the amount of requirement-satisfied coflow changes if coflows have looser deadlines. To this end, we increase x, i.e., the mean value of z, or the so-called scale factor of deadline, from 1 to 5. As Figure 5b reveals, for all schemes but Poco, a significant increased amount of coflows would meet their requirements when their deadlines get relaxed. However, the results of Poco have little change. This is reasonable since the relaxation of deadline would not reduce the network load indeed. As Poco has already made very efficient use of the network to admit the request, there is little room to improve.

Impact of network load. Next, we test the change of requirement-satisfied coflow under various network loads. According to the trace generator, the tested coflow requests are assumed to arrive in a Poisson process whose rate is λ. We vary the network load from 0.6 to 1.2 by controlling the rate parameter λ. Because a coflow will get expired automatically after its deadline, there would be only a limited amount of coflows to server even if the network load runs into a load value larger than 1. As Figure 5c indicates, for all schedule schemes, the percentages would reduce with the increase of network load, consistent with the fact that more requests will get completely served if the network load is light. We also notice that once the network load is under 0.6, Poco would let all requests meet their requirements simultaneously. This reflects that Poco does make very efficient rate allocations.

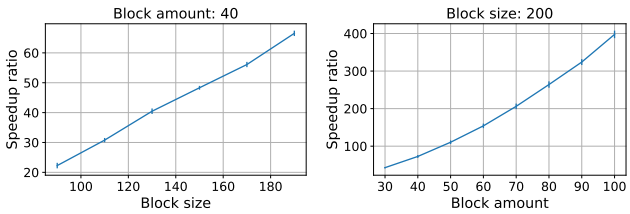
Speedups. Now, we test the efficiency of Poco's core solver. To highlight the speedups of using the block-angle constraint structure while eliminating other effects, we use the straightforward implementation of the solver core shown in Figure 3 to solve randomly generated constraints and right hand vectors on an Ubuntu 18.04 server equipped with one Intel Xeon(R) Silver 4210 CPU and two 8 GB DDR4 memory. In tests, we assume constraint blocks are square and share the same size, then test how the run time changes with the block size and amount (i.e., parameter n in (16)). Basically, the solving time of Poco solver ranges from several milliseconds to several hundreds of milliseconds, depending on the problem size. As is known, the concrete running time could be greatly improved by taking advantage of hardware specific optimizations like MKL.[3] To normalize the effects of hardware, we compare the run time of Poco solver with that of the naive solver unaware of the constraint structure.

[3]Intel@Math Kernel Library, https://software.intel.com/en-us/mkl



(a) Impact of completeness requirement (b) Impact of deadline requirement (c) Impact of network load

Figure 5: Different from the significant performance degradation of FS, Con-Myopic, and Varys, the percentage of coflows that admitted by Poco only decreases slightly with the increase of required completeness and network load. As well, Poco always outperforms all other scheduling algorithms greatly.



(a) Under various block size (b) Under various block amount

Figure 6: Compared with its native implementation, the core computation involved by Poco solver can be greatly accelerated by taking advantage of the specific constrict structure shown in (16)**; and the speedup ratios increase with both the block amounts and block sizes: the speedup ratios increase almost linearly with both the block amounts and block sizes.**

As Figure 6 implies, compared with the naive solver, the Poco solver obtains huge performance gain by making use of the block-angle structure of the constraint matrix: the speedup grows almost linearly with both the size and amount of involved constraint blocks. For instance, the ratio would reach nearly 400 when the constraint matrix involves 100 angle blocks, each of which involves 200 variables and constraints. It is reasonable, since the solver design shown in Figure 3 not only makes the computation parallelable, but also eliminates a lot of useless calculation by using the sparsity of constraints. Because of the limits of hardware capacity, the speedup ratio would not grow without bound. Nevertheless, the results still give us a strong insight that huge performance gain would be achieved by making use of the well-structure of constraints.

6 CONCLUSION

Nowadays, an increasing number of emerging time-sensitive distributed applications are able to tolerate loss-bounded inputs by design [3, 9, 16, 20], yielding novel design space and trade-offs for the schedule of their coflow transmissions. Accordingly, this paper studied this type of trade-off and proposed Poco, a POlicy-based COflow scheduler, to achieve tolerance-aware coflow scheduling based on the requirements of applications. As confirmed by extensive trace-driven simulations, by trading loss-bounded completeness for timeliness and trading one coflow's completeness for those of others on demand, Poco was able to achieve optimal bandwidth allocations respecting user-specific requirements; and, by making use of the constraint structure of the problem, Poco obtained linear speedups for the computation of rate scheduling.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback. The work of Shouxi Luo was supported in part by the China Postdoctoral Science Foundation under Grant 2019M663552, and in part by the Fundamental Research Funds for the Central Universities under Grant 2682019CX61. The work of Pingzhi Fan was supported in part by NSFC Project under Grant 61731017, and in part by the National Key Research and Development Program of China under Grant 2018YFB1801104. The work of Hongfang Yu was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1802800, and in part by the PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications under Grant PCL2018KP001.

REFERENCES

[1] Erling D. Andersen and Knud D. Andersen. 2000. The Mosek Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm. In *High Performance Optimization*. Springer US, 197–232.
[2] Dimitris Bertsimas and John Tsitsiklis. 1997. *Introduction to Linear Optimization* (1st ed.). Athena Scientific.
[3] L. Chen, W. Cui, B. Li, and B. Li. 2016. Optimizing coflow completion times with utility max-min fairness. In *INFOCOM*. 1–9.
[4] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A Networking Abstraction for Cluster Applications. In *11th HotNets* (Redmond, Washington). 31–36.
[5] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient Coflow Scheduling with Varys. In *SIGCOMM* (Chicago, Illinois, USA). 443–454.
[6] Jacek Gondzio and Robert Sarkissian. 2003. Parallel interior-point solver for structured linear programs. *Mathematical Programming* 96, 3 (01 Jun 2003), 561–584.
[7] S. Im, M. Shadloo, and Z. Zheng. 2018. Online Partial Throughput Maximization for Multidimensional Coflow. In *INFOCOM*. 2042–2050.
[8] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. 2014. Calendaring for Wide Area Networks. In *SIGCOMM* (Chicago, Illinois, USA). 515–526.

[9] Ke Liu, Shin-Yeh Tsai, and Yiying Zhang. 2019. ATP: a Datacenter Approximate Transmission Protocol. *preprint arXiv:1901.01632* (2019).

[10] S. Luo, T. Ma, W. Shan, P. Fan, H. Xing, and H. Yu. 2020. Efficient Multi-source Data Delivery in Edge Cloud with Rateless Parallel Push. *IEEE Internet of Things Journal* (2020), 1–1. https://doi.org/10.1109/JIOT.2020.2996800

[11] S. Luo, H. Yu, K. Li, and H. Xing. 2020. Efficient File Dissemination in Data Center Networks With Priority-Based Adaptive Multicast. *IEEE Journal on Selected Areas in Communications* 38, 6 (2020), 1161–1175.

[12] S. Luo, H. Yu, and L. Li. 2016. Decentralized deadline-aware coflow scheduling for datacenter networks. In *IEEE ICC*. 1–6.

[13] S. Ma, J. Jiang, B. Li, and B. Li. 2016. Chronos: Meeting coflow deadlines in data center networks. In *IEEE ICC*. 1–6.

[14] Y. Peng, K. Chen, G. Wang, W. Bai, Y. Zhao, H. Wang, Y. Geng, Z. Ma, and L. Gu. 2016. Towards Comprehensive Traffic Forecasting in Cloud Computing: Design and Application. *IEEE/ACM Transactions on Networking* 24, 4 (Aug 2016), 2210–2222.

[15] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM* (Chicago, Illinois, USA). 307–318.

[16] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding. In *OSDI* (Savannah, GA, USA). 401–417.

[17] S. M. Srinivasan, T. Truong-Huu, and M. Gurusamy. 2018. Deadline-Aware Scheduling and Flexible Bandwidth Allocation for Big-Data Transfers. *IEEE Access* 6 (2018), 74400–74415.

[18] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A Survey on Distributed Machine Learning. *ACM Comput. Surv.* 53, 2, Article 30 (March 2020), 33 pages. https://doi.org/10.1145/3377454

[19] H. Wang, L. Chen, K. Chen, Z. Li, Y. Zhang, H. Guan, Z. Qi, D. Li, and Y. Geng. 2015. FLOWPROPHET: Generic and Accurate Traffic Prediction for Data-Parallel Cluster Computing. In *35th IEEE ICDCS*. 349–358.

[20] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. 2019. Rethinking Transport Layer Design for Distributed Machine Learning. In *3rd APNet* (Beijing, China). 22–28.

[21] Chen Yu, Hanlin Tang, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ce Zhang, and Ji Liu. 2019. Distributed Learning over Unreliable Networks. In *36th ICML*, Vol. 97. 7202–7212.