Courier: A Unified Communication Agent to Support Concurrent Flow Scheduling in Cluster Computing

Zhaochen Zhang, Xu Zhang, Zhaoxiang Bao, Liang Wei,
Chaohong Tan, Wanchun Dou, Guihai Chen, Chen Tian

Abstract—As one of the pillars in cluster computing frameworks, coflow scheduling algorithms can effectively shorten the network transmission time of cluster computing jobs, thus reducing the job completion times and improving the execution performance. However, most of existing coflow scheduling algorithms failed to consider the influences of concurrent flows, which can degrade their performance under a massive number of concurrent flows. To fill the gap, we propose a unified communication agent named Courier to minimize the number of concurrent flows in cluster computing applications, which is compatible with the mainstream coflow scheduling approaches. To maintain the scheduling order given by the scheduling algorithms, Courier merges multiple flows between each pair of hosts into a unified flow, and determines its order based on that of origin flows. In addition, in order to adapt to various types of topologies, Courier introduces a control mechanism to adjust the number of flows while maintaining the scheduling order. Extensive large-scale trace-driven simulations have shown that Courier is compatible with existing scheduling algorithms, and outperforms the state-of-the-art approaches by about 30% under a variety of workloads and topologies.

Index Terms—Data center network, coflow scheduling, congestion control.

1 Introduction

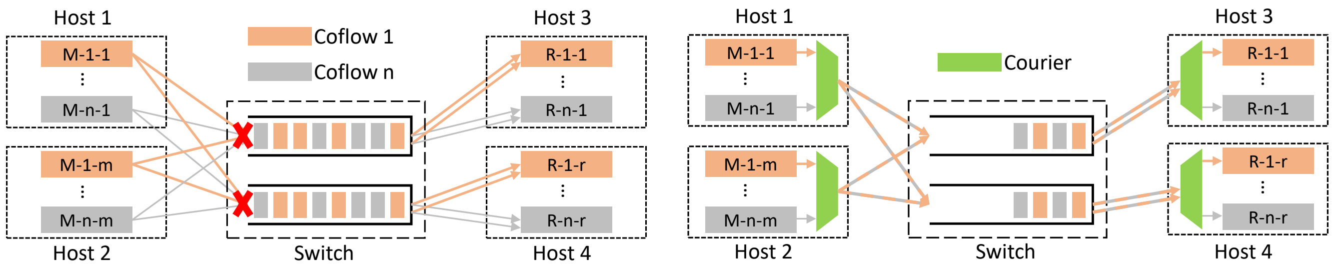
Cluster computing frameworks have been widely deployed in data centers [1], [2] due to their high throughput and low cost for processing large amounts of data. Job completion time (JCT) is the critical metric of execution performance for a cluster computing job. Recent studies have shown that the communication stage, which takes place between groups of machines in successive computation stages, significantly impacts the JCT in cluster computing, affecting both traditional MapReduce jobs [3] and emerging distributed DNN (Deep Neural Networks) training jobs [4], [5]. Typically, the next computation stage cannot begin until all flows within the communication stage are completed. To capture this all-or-nothing semantic, the concept of coflow [6] was introduced, defined as the set of all flows within an all-or-nothing communication stage.

Existing solutions and their limitations. To reduce the JCT of cluster computing jobs, extensive scheduling algorithms have been proposed, which can be divided into two categories: **rate-based** scheduling algorithms and **order-based** scheduling algorithms. Rate-based scheduling algorithms [3], [7], [8], [9], such as Aalo [8], schedule by assigning rates to flows. In contrast, order-based scheduling algorithms [10], [11], such as Sincronia [10], prioritize scheduling by ensuring that higher-order coflows complete before lower-priority ones. However, most of existing general scheduling algorithms neglect the influences of concurrent flows (concurrent flow issue for short)(§ 2.1). Fig. 1(a) illustrates the coflows in Hadoop, where mappers and reducers connect in a fully connected manner, meaning that the number of flows in a coflow is the product of the number of mappers and the number of reducers. Without limiting the number of concurrent flows, data center networks can experience packet loss rates up to 2%, resulting in an approximately $1.5\times$ increase in CCT (§ 5.1).

Attempts to address the concurrent flow issue have been made through cluster computing frameworks and concurrent flow-oriented scheduling algorithms, though each approach exhibits its own limitations. For example, Hadoop [12] addresses the concurrent flows issue by limiting the number of concurrent flows each reducer can handle, which is an adjustable parameter. However, determining the optimal number of concurrent flows per reducer is challenging, as both too few and too many flows can adversely affect the completion time (CCT). In addition, Hadoop's solution will undermine existing coflow scheduling algorithms, thus reducing the overall performance. Django is a state-of-the-art coflow scheduling algorithm that considers the concurrent flow issue. It utilizes a Support Vector Machine (SVM) to predict the optimal number of concurrent flows, then employs a centralized scheduler to schedule coflow within limits. The primary limitation of Django is its scalability, which is hindered by its dependency on a coordinator and its tendency to block small coflows (§ 2.4).

Our Contributions. To bridge the gap, we propose Courier, a unified communication agent to minimize the number of concurrent flows in cluster computing applications. Courier's key insight is to merge concurrent data flows with identical source-destination pairs, thereby reducing the number of flow without sacrificing transmission opportuni-

- *Z. Zhang, Z. Bao, W. Dou, G. Chen, C. Tian are with the State Key Laboratory for Novel Software Technology, Nanjing University.*
- *X. Zhang was with the School of Electronic Science and Engineering, Nanjing University.*
- *L. Wei, C. Tan are with Jiangsu Future Network Innovation Institute.*
- *X. Zhang and C. Tian are corresponding authors of this paper. Email: xzhang24.nju@gmail.com, tianchen@nju.edu.cn.*



(a) The cluster computing jobs build the queue on the switch.

(b) Courier reduces the number of flows.

Fig. 1. The insight of Courier. The rectangle on the host represents a cluster computing task; the first letter **M or R** indicates the task is a mapper or a reducer; the second number **from 1 to n** indicates which job the task belongs to, and the third number **from 1 to m (or r)** indicates the order of this task in the job.

ties. As shown in figure 1(b), we deploy Courier on each server. By merging multiple flows between each pair of hosts into a single flow, the number of concurrent flows is significantly reduced in the data center network.

There are two challenges for the design of Courier. The first challenge is how to maintain the scheduling order given by the scheduling algorithms while merging flows. There are various coflow scheduling algorithms that use different underlying mechanisms to enforce the scheduling order. We need an elaborate approach to merge flows that use different underlying scheduling enforcement mechanisms. The second challenge is that rigidly merging flows between each source-destination pair into one flow struggles to cope with complex data center topologies. For example, in a data center with a large topology, maintaining even just one flow per pair of servers can lead to an excessive number of flows in the network, which results in the failure of services due to the incurred high JCT. And in a data center with a multi-path topology, Courier cannot take advantage of the multiple paths between each pair of servers as the number of flows is only one.

To overcome the first challenge, we design flow merging approaches for two major kinds of coflow scheduling algorithms separately(§ 4.1). To be compatible with those rate-based scheduling algorithms, Courier sets the unified flow's rate to the sum of each flows started by reducers and divides the transferred data based on the original rate among the reducers. To be compatible with those order-based scheduling algorithms, Courier sets the priority of the unified flow to the highest one among the flows and divides the transferred data among the reducers with the highest priority. The mechanisms guarantee Courier is compatible with existing coflow scheduling algorithms. By combining with Courier, existing coflow scheduling algorithms can mitigate the influence of queue buildup incurred by the concurrent flows.

To tackle the second challenge, a flexible flow number control mechanism is elaborated for Courier. To avoid the concurrent flow problem in the large topology, Courier uses a scheduling-friendly flow number limitation mechanism that starts important flows first. With this mechanism, Courier guarantees the scheduling order while limiting the number of flows (§ 4.2.1). For the multi-path topology, Courier increases the number of flows between each pair of servers to take full advantage of the topology (§ 4.2.2).

We evaluate Courier through large-scale trace-driven simulations on NS-3 (§ 5). The results show that Courier can effectively reduce the number of concurrent flows and avoid packet loss in most scenarios. Under MapReduce workloads, Courier is able to achieve at least 15% improvements in coflow performance. With the flexible flow number control mechanism, Courier can reduce CCT and packet loss in a variety of topologies. Aalo and Sincronia have 16% and 29% improvement on average over all workloads by combining with Courier, respectively. We also conduct experiments comparing Courier with Django, a state-of-the-art scheduling-based flow number control method, and the results show that the combination of Courier with mainstream scheduling algorithms has about 30% higher optimization on CCT than Django. In addition, we validate the performance improvements of Courier in the emerging distributed DNN training scenario. The results show that Courier can deliver up to a 26% improvement in average CCT and up to a 35% improvement in tail CCT in this scenario.

The major contributions of our paper are summarized as follows:

- We propose Courier, a unified communication agent in cluster computing frameworks. Courier mitigates the concurrent flow problem by merging multiple flows between each pair of servers.
- We analyze and reduce the underlying mechanism of common coflow scheduling algorithms and make Courier compatible with them. A flexible flow number control mechanism is elaborated to make Courier compatible with various topologies.
- We evaluate Courier through large-scale trace-driven simulations on NS-3. Experiments have shown that Courier outperforms the state-of-the-art approaches in terms of network performance, even under different types of workloads and network topologies.

The rest of the paper is organized as follows. We illustrate the background and motivation of Courier in Section 2. The overview of Courier is presented in Section 3. We detail the design of Courier in Section 4. We demonstrate the evaluation of Courier in Section 5. Then we discuss Courier's benefits on other congestion control protocols and coflow scheduling algorithms in Section 6. Finally, we draw the conclusion in Section 7.

2 Background and Motivation

In this section, we analyze the background and motivation for Courier.

2.1 Cluster Computing Job and Coflow

Cluster computing, such as MapReduce and distributed DNN training have been widely employed in data centers [1], [2], [5] due to its high throughput and low cost for processing large amounts of data. Recent studies have shown that network transmission significantly impacts the JCT in cluster computing [3], [4], [5]. Fortunately, the structured procedure of cluster computing jobs makes it possible to optimize network transmission to reduce their JCTs. In this paper, we primarily focus on the representative computing framework of MapReduce, Hadoop [12]. Our work is also applicable to distributed DNN training [1], which is detailed in § 5.4.

In Hadoop, the data is first processed by m mappers deployed on some servers and then by r reducers deployed on other servers. The reducers will start only after all intermediate data have been successfully transferred. Thus the intermediate data communication stage contains $m*r$ flows between mappers and reducers. Typically, the computation stage cannot start until all flows within the communication stage finish. To capture this all-or-nothing semantic, coflow [6] is introduced and **defined as all flows within an all-or-nothing transmission phase.**

Relying on coflow abstraction, many practical coflow scheduling algorithms emerged. Based on their underlying mechanisms to schedule flows, they can be divided into **rate-based** and **order-based** scheduling algorithms. Varys [7] uses the Smallest-Effective-Bottleneck-First heuristic to order coflows and then **assigns rates** to *all flows* based on the order and predicted CCT of its coflow. Aalo [8] separate coflows into several priority queues, and the priority of a coflow is determined in the least attained service (LAS) discipline. Then, Aalo **assigns rates** to *all flows* in a max-min fairness manner. Sincronia [10] periodically calculates the orders of coflows by a greedy algorithm. Unlike previous algorithms, Sincronia **assigns orders** to *all flow*, thus offloading rate allocation and order guarantees to the underlying priority-enabled protocol, *e.g.*, IP protocol with differentiated services code point (DSCP). In contrast to other algorithms that focus on the rate of the flows, Sincronia **focuses only on the completion order of the flows** (*i.e.*, higher order flows complete before lower order ones), so it is called the order-based algorithm. Although these algorithms have been shown to be practical in data center networks, they invariably start as many flows as possible, ignoring the concurrent flow problem.

In addition to the above algorithms, many scheduling algorithms focus on theoretical analysis. The common problem of theoretical works is that they model the data center network as an ideal graph and ignore the realistic details of network devices. For example, Qiu *et al*. [13], Khuller *et al*. [14], Shafiee *et al*. [15], and Ahmadi *et al*. [16] model the data center network as a big non-blocking switch without consideration of buffer. Jahanjou *et al*. [17] and Chowdhury *et al*. [18] model the data center network as a directed graph, where the vertices are servers and the edges are links with fixed capacity, without considering the buffer of the switches. All theoretical algorithms are rate-based algorithms and ignore the concurrent flow problem as they disregard the realistic details of network devices. In conclusion, most of existing coflow scheduling algorithms neglect the influences of concurrent flow.

2.2 Unavoidable Queuing and Packet Loss

It is important to minimize queuing and the potential packet loss due to excessive queuing in the data center network. To this end, DCTCP, a variant of TCP, is widely deployed in the data center. DCTCP is quite efficient at reducing the queue length on switches while maintaining the same throughput as the original TCP. However, DCTCP is insufficient to solve the concurrent flow problem caused by cluster computing jobs. Both analysis [19] and experiments [20] show that there is a linear relationship between the max queue length and the number of concurrent flows in the steady state:

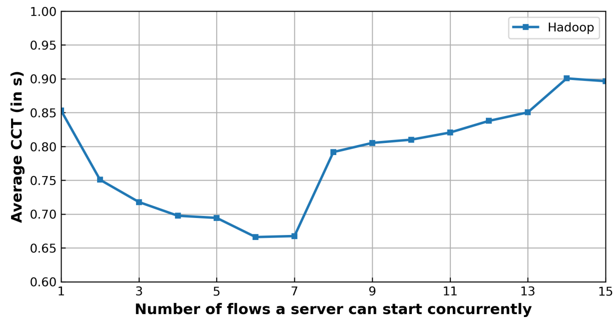
$$q_{max} = k + n \tag{1}$$

Where q_{max} is the max length of the queue in the switch, k is the explicit congestion notification (ECN) marking threshold, and n is the number of concurrent flows. When the number of concurrent flows increases, the queue length inevitably increases under DCTCP, affecting latency-sensitive flows in the data center on the one hand, and even causing packet loss on the other.

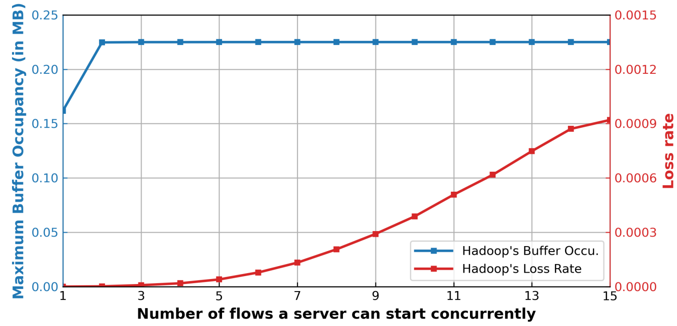
To illustrate the extent and impact of packet loss in the cluster computing frameworks, we replayed the production trace from Facebook [7] with Hadoop's mechanism under DCTCP in the NS-3 simulator (Fig. 2 and Fig. 3). In the simulation, the buffer size for each port is set to 0.225 MB, consistent with the configuration used in DCTCP [19]. As shown in Fig. 2, the independent variable is the number of flows a reducer can start concurrently, which is proportional to the maximum number of flows in the network. As shown in Fig.2(a), when the number of flows is small, CCT increases because bandwidth is not fully utilized. And when the number of flows is large, the increase in packet loss rate also leads to an increase in CCT. Fig.2(b) shows the variation of maximum buffer occupancy and loss rate in the experiment. When the number of flows is 1, DCTCP effectively controls the queue length and no packet loss occurs. When the number of flows increases, the switch's buffer is exhausted and the loss rate gradually rises.

Packet loss not only wastes bandwidth but also significantly prolongs FCT due to timeouts, which in turn affects CCT. Fig. 3 illustrates the network performance under different retransmission timeout (RTO) settings. It can be observed that RTO has a significant impact on the CCT and the loss rate. The analysis of the experiments shows that about 45% of coflow's last completed flow has experienced timeout, even though the packet loss rate was only 0.6%. In other words, if we can reduce the impact of the timeout, nearly half of the coflows' completion time can be improved.

1. Our work is also applicable to workloads where barriers exist between communication and computation phases (*i.e.*, computations require to waiting until all communications are complete), such as in Bulk Synchronous Parallel (BSP).

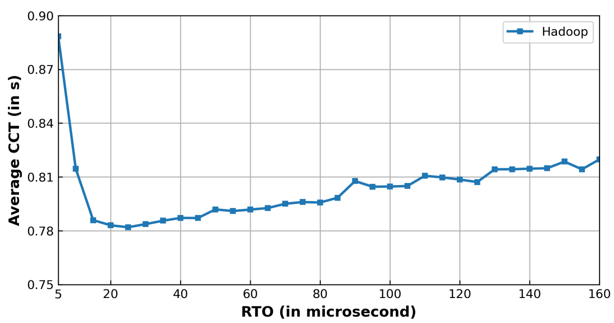


(a) Average CCT.

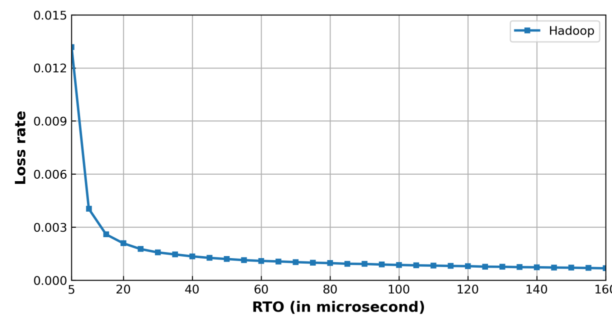


(b) Buffer Occupancy & Loss Rate.

Fig. 2. Brief evaluation of Hadoop under DCTCP.



(a) Average CCT.



(b) Loss rate.

Fig. 3. Impact of the RTO of Hadoop under DCTCP.

There are many efforts to reduce the impact of timeouts on network performance. Tail loss probe (TLP) [21] uses a shorter probe timeout (PTO) to reduce the impact of the RTO on the flow, which requires extending the TCP protocol and does not completely avoid the impact of timeouts. Priority-based Flow Control (PFC) [22] avoids timeouts by making the network lossless, but it brings problems such as head-of-line blocking and deadlocks. There are also many studies dedicated to providing fast loss notifications to avoid timeouts [23], [24], [25], but they all require programmable switches or modifications to the switch chip. Among all the ways trying to mitigate the impact of timeouts, the most practical one is setting the appropriate RTO [26]. However, the experiments shown in Fig. 3 have already demonstrated that the appropriate RTO is difficult to choose. With an aggressive RTO, the network is overflowed due to spurious retransmissions (Fig. 3(b)), and the CCT suffers as a result (Fig. 3(a)). But a conservative RTO also prolongs CCT when the timeout occurs. To the best of our knowledge, no existing work can significantly reduce the impact of packet loss without protocol modification, dedicated equipment, or elaborate parameter setting.

2.3 Concurrent flow issue is exacerbating

The trend of placing an increasing number of cores on a single server is exacerbating the concurrent flow issue. To illustrate this trend, we conduct a survey of server-level chips and mainstream cloud service providers. All the data is taken from the official websites of these companies. Table 2 shows the increase in the number of cores in top-level processors from 2017 to 2021. Table 2 lists the maximum number of cores in bare metal servers provided by mainstream cloud server providers, which may reflect the core number of current mainstream data center servers. It is worth noting that all of these servers support hyper-threading, which makes one physical core works as two or more logical cores. For example, the server ”o2-ultramem-896-metal” will have 896 threads when hyper-threading is enabled.

Housing more cores in a single server enhances its processing performance. While gaining higher performance,

TABLE 1
Number of cores of server-level processors from 2017 to 2021

Processer provider		2017	2019	2021
Intel	Max.	28	56	40
	Avg.	23	28	35
AMD	Max.	32	64	64
	Avg.	22	31	33

TABLE 2
Number of cores of bare metal servers provided to common users

Cloud server provider	Machine type	# of cores
AWS	i3.metal	36
Google Cloud	o2-ultramem-896-metal	448
Alibaba Cloud	ecs.ebmhfc7.48xlarge	96
Huawei Cloud	physical.ks1.2xlarge	64

we are also putting more pressure on the network. The more threads a server supports, the more tasks run on the server. As mentioned in § 2.1, the number of flows is positively correlated with the number of tasks. Therefore, the trend of increasing core number will eventually lead to more and more buffer pressure on the switch. Over time, concurrent flow problems in our data center networks will worsen.

2.4 Existing Solutions

To address the concurrent flow problem, Hadoop adopts a workaround solution that restricts the number of flows a reducer can start simultaneously, which is an adjustable parameter. However, this solution has three drawbacks. (i) The optimal number of flows per reducer is difficult to determine. As shown in Fig. 2, on the one hand, we cannot arbitrarily reduce the number of concurrent flows in Hadoop. Otherwise, the CCT is prolonged due to bandwidth underutilizing. On the other hand, too many concurrent flows are also harmful to CCT, as discussed in § 2.2. (ii) The limitation of the number of concurrent flows will affects the effectiveness of the coflow scheduling algorithms. (iii) Moreover, Hadoop's approach does not take into account the number of reducers on a server and will fail to alleviate the concurrent flow problem as discussed in § 2.3.

Django is the state-of-art work that attempts to solve the concurrent flow problem in the field of coflow scheduling. Django first uses a Support Vector Machine (SVM) to predict the optimal number of concurrent flows. Then, Django uses a coflow scheduling algorithm with a centralized coordinator to limit the number of flows.

Django, however, has some drawbacks. (i) The implementation of flow number restriction relies on a coordinator, which makes Django lack scalability. (ii) The scheduling algorithm of Django takes some time to take effect; thus, small coflows will be blocked. (iii) The predicted flow number is highly related to the network environment, causing Django to be sensitive to changes in the network environment (link failures, device maintenance, *etc*.).

3 Overview

3.1 Key Idea

The design of Courier can be illustrated in figure 1. Suppose there are n coflows each with m mappers and r reducers deployed on four hosts, respectively. In the origin case shown in figure 1(a), mappers and reducers communicate in a fully connected manner. As a result, there are $n * m * r$ flows in the network, putting a considerable buffer pressure on the switch.

However, as figure 1(b) shows, we can reduce the number of flows to 4 with Courier, which is deployed on each host. When mappers and reducers have communication requirements, they interact with Courier instead of starting flows themselves. Then Courier will start unified flows to Couriers on other hosts it needs to communicate with. When the intermediate data transfer is completed, the Courier on the reducer side distributes the data to each reducer separately.

By using the mechanism declared above, **Courier reduces the number of flows between a pair of servers from n to 1.** We then illustrate how Courier solves problems described in § 2.

Concurrent flow problem in cluster computing and coflow scheduling (§ 2.1). With Courier, the number of flows in the cluster computing framework basically does not increase with the number of coflows n, the number of mappers m and reducers r per coflow. In other words, the number of flows in the network is reduced from $O(n*m*r)$ to $O(1)$, significantly alleviating the concurrent flow problem. Courier focuses only on the aggregation of flows but not on coflow scheduling, so it is orthogonal to existing coflow scheduling algorithms. By integrating with Courier, coflow scheduling algorithms can escape the concurrent flow problem.

Unavoidable queuing and packet loss (§ 2.2). By merging flows, Courier can drastically reduce the number of concurrent flows in the cluster computing framework. This means that the steady-state queue length on the switch will be significantly reduced under DCTCP. Shorter queue length on the one hand facilitates latency-sensitive flows in the data center, and on the other hand indicates a lower probability of packet loss. Therefore, the probability of timeout is significantly reduced, and there is no need to use additional mechanisms to mitigate the impact of timeout on CCT. In addition, Courier aggregates multiple small flows into a large flow, which facilitates the congestion control protocol to take effect and mitigate the risk of incast [27], [28], [29].

The trend of housing more cores on a server (§ 2.3). Courier fundamentally decouples the number of concurrent flows and the number of reducers in a single server. By using Courier, data center managers can upgrade servers (*i.e.* increase the number of cores) without worrying about the network problems it might bring.

Courier outperforms existing solutions that limit the number of flows (§ 2.4) in two ways. First, Courier does not incur head-of-queue blocking, i.e., the reducers can start all the flows they need and thus complete the data transmission as fast as possible. Second, Courier takes effect at server granularity, with neither the potential problems of reducer granularity solution (Hadoop) nor the requirement for a centralized coordinator (Django).

3.2 Challenges

There exist many design challenges to be addressed to make Courier practical in data center networks:

Integration with coflow scheduling algorithms. There are many successful coflow scheduling algorithms. With these coflow scheduling algorithms, the average CCT can be significantly reduced, and cluster computing jobs can be accelerated. But different coflow scheduling algorithms require various underlying scheduling mechanisms. It is a huge challenge for Courier to meet the requirements of the scheduling algorithms while merging multiple flows into one.

Coping with large topologies. Courier ensures at most one flow between each pair of servers, thus minimizing the buffer pressure on the switches. However, if the network topology of the data center is particularly large (e.g., thousands of servers or more), the number of one-to-one

Algorithm 1: Design of Courier

```
Input: S: Interface of deployed scheduling algorithm
       MaxNum: The maximum number of flow
       MinSize: The minimum size of flow
1  Procedure Main(S, MaxNum, MinSize)
2      L ← [ ], L_old ← [ ], T ← S.type()
3      while true do
4          L ← S.schedule()
5          if L ≠ L_old then
               /* The scheduling result changes */
6              L_old ← L
7              L_uni ← Merge(L, T)
                                   // Merge the flows
8              L_act ← SelectAndSplit(L_uni, T,
9                      MaxNum, MinSize)
                              // Decide flows to start
10             for f_act in L_act do Start flow f_act
                               // Actually start flows
11     end
```

Algorithm 2: Merge Flows into Unified Flows

```
Input: L: The list of scheduled flows
       T: The type of the scheduling algorithm
Output: L_uni: The list of unified flows.
1  Procedure Merge(L, T)
2      L_uni ← [ ]                 // List of unified flows
3      for f in L do
4          if L_uni.contain(f.addr) = false then
5              L_uni[f.addr] ← f
6          else
7              L_uni[f.addr] ←
                 MergeFlow(L_uni[f.addr], f, T)
8      end
9      return L_uni
10 Procedure MergeFlow(f_1, f_2, T)
11     f ← < f_1.addr, 0, 0, 0 >
                            // Note that f_1.addr = f_2.addr
12     if T = "rate-based" then
13         f.rate ← f_1.rate + f_2.rate
14         f.size ← f_1.size + f_2.size
15     if T = "order-based" then
16         if f_1.order = f_2.order then
17             f.order ← f_1.order
18             f.size ← f_1.size + f_2.size
19         else
20             f ← the flow with the higher order bewteen
                 f_1 and f_2
21     return f
```

connections between these servers will become very large either.

Leveraging multipath topology. Nowadays, most data centers adopt a dense interconnect structure [30], [31], [32] to achieve higher aggregate bandwidth and robustness [33]. In other words, there are many alternative paths between certain pairs of hosts. However, as there is only one flow between each pair of servers, Courier cannot directly utilize multipath topology.

4 DESIGN

This section details the design of Courier through showing how to solve the challenges. The general design of Courier is presented in Alg. 1. Courier takes the interface of coflow scheduling algorithm S, the maximum number of flows can be started concurrently $MaxNum$ and the minimum size of a flow $MinSize$ as input. The interface S needs to expose two methods: $S.type()$, which returns the type (rate-based or order-based) of the scheduling algorithm, and $S.schedule()$, which returns a list L of scheduled flows. A flow f is define as four-tuple $< f.addr, f.size, f.rate, f.order >$. The first element of the four-tuple represents the source-destination IP address pair of the flow, the second element represents the size of the flow, the third element represents the flow rate in the rate-based scheduling algorithm, and the fourth element is the order of the flow in the order-based algorithm. Courier polls the scheduling algorithm (lines 3-4 in Alg. 1), and if the result given by the scheduling algorithm changes(lines 5-6), the flows actually started by Courier are recalculated (lines 7-8) and restarted (line 9).

The detailed process of Courier is divided into two parts. First, Courier merges the flows given by the scheduling algorithm into unified flows, as described in Alg. 2 (turns L into L_{uni}, which is the list of *unified* flows). Flows with the same source-destination address pair are merged into a unified flow in this part. Then Courier decides which flows to start based on a pre-specified maximum number of flows $MaxNum$, as described in Alg. 3 (turns L_{uni} into L_{act}, which is the list of the flows that are *actually* started). This section details how Courier merges flows (§ 4.1) and how Courier derives actual flows to send (§ 4.2).

4.1 Compatibility with Scheduling Algorithms

Coflow scheduling has attracted extensive attention from academia and industry. For instance, Aalo schedules by controlling the sending rate of the flows. Sincronia assigns different priorities to coflows and relies on the DSCP to ensure that high-order coflows are completed first. Nevertheless, how to guarantee the correctness of scheduling order when merging multiple flows into one is a crucial issue. Courier's merge mechanism is demonstrated in Alg. 2, which takes the list of scheduled flows L and the type of the scheduling algorithm T as input and returns a list of merged flows L_{uni}. As shown in lines 3-8 of Alg. 2, Courier merges flows by source-destination address pair. The detailed operation of merging two scheduled flows is related to the type of scheduling algorithm (lines 10-21 of Alg. 2).

The majority of scheduling algorithms are **rate-based** algorithms [3], [7], [8], [9], [13], [14], [15], [16], [17], [18]. The scheduling algorithms allocate the sending rate of each flow through a modified kernel module or user-space network stack. Flows with high priorities are often assigned with a higher rate in order to make them complete quickly. To prevent starvation, flows with lower priorities are also assigned with a lower rate rather than not being transmitted at all. In this case, we consider n flows with the same source-destination address pair, whose origin rate is R_{f_i} and size is S_{f_i}. These flows are merged into a unified flow f_{uni} by Courier. The rate $R_{f_{uni}}$ and data size to transfer $S_{f_{uni}}$ of

f_{uni} can be described using the following equations:

$$R_{f_{uni}} = \sum_{i=1}^{n} R_{f_i},\ S_{f_{uni}} = \sum_{i=1}^{n} S_{f_i} \tag{2}$$

As described in equation 2, Courier simply sets the sending rate (size) of the unified flow to the sum of all the merged flows' rates (size). The flow merging mechanism of the rate-based scheduling algorithm is described in the Alg. 2 by lines 12-14. Using this simple merging method, Courier can be integrated with rate-based coflow scheduling algorithms.

The **order-based** scheduling is first proposed by Sincronia [10] and has been widely studied now [34]. In contrast to the rate-based mechanism, the order-based mechanism only requires "order-preserving" — if coflow A has a higher order than coflow B, flows in A must hold an order higher than flows in B. This mechanism is often implemented using a priority-enabled network protocol stack (*e.g.*, Internet Protocol with DSCP). The order-based flow merging mechanism is described in the Alg. 2 by lines 15-21. If the two flows to be merged have the same order, Courier merges them by adding their sizes up (lines 16-18 of Alg. 2). If the two flows have different orders, Courier directly selects the higher-order flow to transmit as the unified flow (lines 19-20 of Alg. 2). It is worth noting that if the switch in the data center network allocates separate buffer space for each priority, the unified flow's window needs to be reset to prevent buffer overflow when changing the priority.

Based on the above-described principles, coflow scheduling algorithms that rely on other mechanisms can be easily integrated with Courier. For example, a scheduling algorithm that only controls whether a flow is started or not can be directly reduced to an order-based scheduling algorithm that has only one priority.

4.2 Compatibility with Various Topologies

The topology of real data centers is often large and complex. In such topologies, it can be harmful to rigidly maintain exactly one flow between each pair of servers. In the large topology, even only one flow between each pair of servers can lead to an excessive number of concurrent flows. In contrast, in the multipath topology, only one flow between each pair of servers cannot fully utilize the network bandwidth. Actually, most data centers contain thousands of servers and use a multipath topology simultaneously, in which case determining the number of flows between pairs of servers becomes a dilemma.

To solve this problem, after merging flows, Courier adaptively decides which flows to start or increases the number of flows between each server pairs through Algorithm 3. The algorithm determines up to $MaxNum$ flows to actually start (L_{act}) from unified flows (L_{uni}). It first **selects** unified flows from important (higher rate or priority) to unimportant into L_{act} if the number of flows in L_{act} is less than $MaxNum$ and there are remaining flows in L_{uni} (lines 3-10, § 4.2.1). If the number of flows in L_{act} remains below $MaxNum$ after incorporating all flows in L_{uni}, Courier will repeatedly **split** the most important flow in L_{act} into two flows until $MaxNum$ is reached (lines 11-17, § 4.2.2). The overall rationale of the algorithm and the selection of $MaxNum$ are discussed in § 4.2.3.

Algorithm 3: Determine Flows to Start

Input: L_{uni}: The list of unified flows
T: The type of the scheduling algorithm
$MaxNum$: The maximum number of flow
$MinSize$: The minimum size of flow
Output: L_{act}: The list of the flows that will actually be started

1 **Procedure** SelectAndSplit($L_{uni}, T, MaxNum, MinSize$)
2 $L_{act} \leftarrow [\,]$ // List of actually start flows
3 **while** $|L_{act}| < MaxNum$ **do**
4 **if** $|L_{uni}| > 0$ **then**
/* Start unified flows with the quantity below $MaxNum$ (§ 4.2.1) */
5 $f_{uni} \leftarrow < 0, 0, 0, 0 >$
6 **if** $T =$*"rate-based"* **then**
7 Select f_{uni} with highest $rate$ from L_{uni}
8 **if** $T =$*"order-based"* **then**
9 Select f_{uni} with highest $order$ from L_{uni}
10 Move f_{uni} from L_{uni} to L_{act}
11 **else**
/* Split flows to utilize multipath topologies (§ 4.2.2) */
12 $f_{act} \leftarrow < 0, 0, 0, 0 >$
13 **if** $T =$*"rate-based"* **then**
14 Select f_{act} with the highest $rate$ and $size$ bigger than $MinSize$ from L_{act}
15 **if** $T =$*"order-based"* **then**
16 Select f_{act} with the highest $order$ and $size$ bigger than $MinSize$ from L_{act}
17 Replace f_{act} with SplitFlow(f_{act}, T) in L_{act}
18 **end**
19 **return** L_{act}
20 **Procedure** SplitFlow(f, T)
21 $f_1 \leftarrow < 0, 0, 0, 0 >, f_2 \leftarrow < 0, 0, 0, 0 >$
22 **if** $T =$ *"rate-based"* **then**
23 $f_1 \leftarrow < f.addr, f.size/2, f.rate/2, 0 >$
24 $f_2 \leftarrow < f.addr, f.size/2, f.rate/2, 0 >$
25 **if** $T =$ *"order-based"* **then**
26 $f_1 \leftarrow < f.addr, f.size/2, 0, f.order >$
27 $f_2 \leftarrow < f.addr, f.size/2, 0, f.order >$
28 **return** f_1, f_2

4.2.1 Compatibility with Large Topologies

To solve the potential concurrent flow problem in the large topology, Courier restricts the number of flows that a single server can start. The flow number restriction mechanism is described by lines 3-10 of Alg. 3. The maximum number of flows that can be started on a single server $MaxNum$ is specified by the data center operators. When the number of flows started by Courier reaches the upper limit, new communication requirements will be delayed instead of establishing connections immediately. A greedy flow selection strategy is adopted to make the flow number restriction mechanism compatible with coflow scheduling algorithms. Courier greedily selects the unified flow with the highest transmission rate or priority as the next flow to be started each time (lines 6-10 of Alg. 3) until the maximum number of flows is reached (*i.e.*, $|L_{act}| = MaxNum$). In case no scheduling algorithm is deployed, Courier can directly select the flow with the biggest transmission size as the next transfer flow each time to minimize head-of-line blocking.

The flow number restriction approach Courier uses is somewhat similar to the workaround approach of Hadoop.

But actually, Courier's approach has many advantages over Hadoop's. (i) Courier restricts the number of flows at the server granularity, but Hadoop restricts it at reducer granularity, which may overload the network with too many reducers(§ 2.3). (ii) Hadoop's scheduling-insensitive approach may hinder the normal functioning of scheduling algorithms. As a comparison, Courier's approach is scheduling friendly because it selects the flows with the highest rate or priority to transmit early.

4.2.2 Utilization of Multipath Topology

Many approaches are widely used in data centers to make full use of the multipath topology. Equal-cost multipath routing [35] (ECMP) is a routing strategy extensively deployed on commercial switches, which randomly hashes flows to equal-cost paths to utilize the bandwidth and balance the load. XPath [36] is a representative method of explicit path control, which explicitly specifies the path for each flow at the host-end.

Courier leverages these well-established multipath protocols to utilize the multipath topology. To this end, Courier increases the number of flows between each pair of servers, and the flows will obey the original multipath routing protocol deployed in the data center. Since the flows between each pair of servers are merged into a single unified flow, the increase in the number of flows is achieved by splitting the unified flow. As in § 4.2.1, Courier uses a similar greedy strategy in order to be compatible with the scheduling algorithm when splitting unified flows. Courier greedily selects the flow with the highest rate or order and size bigger than $MinSize$ (selects the flow with the largest size if rate or order is identical) for splitting until the upper limit of flow number $MaxNum$ is reached (lines 3, 11-17 of Alg. 3). The limit of minimum size $MinSize$ is to prevent Courier from needlessly splitting the flow with a higher rate or order but small size, resulting in a waste of resources. For rate-based scheduling, the rate and size of the actual flow are evenly split between the two flows; for order-based scheduling, the size of the actual flow is evenly divided between the two flows, but the priority remains the same (lines 20-28 of Alg. 3). If no scheduling algorithm is deployed, Courier directly selects the flow with the largest size as the next splitting flow to maximize the utilization of multipath topology.

The number of flows between each pair of servers has to be determined by the deployed multipath mechanism and the network state. In XPath-like mechanisms, the multipath topology can be fully utilized when the number of flows between each server pair is equal to the number of paths, as XPath-like mechanisms ensure that these paths do not cross. Therefore Courier must limit the number of flows between each pair of servers to no more than the number of paths (*i.e.*, limit the number of times a unified flow is split to no more than the number of paths minus one) under XPath-like mechanisms. However, such a rule does not hold in ECMP. Considering the random nature of ECMP, the more the number of flows, the better the utilization of the multipath topology.

4.2.3 Solution to the Dilemma

Courier may face a dilemma in a large multipath topology: how to determine the number of flows between a pair of servers. If the number of flows between a pair of servers is less than or equal to one, the multipath topology cannot be utilized, and more than one may cause the concurrent flow problem.

To address this dilemma, Courier draws on the idea of **max-min fairness**. As illustrated by § 4.2.1 and lines 3-10 of Alg. 3, Courier first incorporates unified flows from important (higher rate or priority) to unimportant into the actual start list L_{act} as much as allowed by $MaxNum$, *i.e.*, the maximum number of flows a server can start. Then, as illustrated by § 4.2.2 and lines 11-17 of Alg. 3, if there are any additional start quotas after all unified flows are in L_{act}, Courier selects the most important flow in L_{act} and splits it to speed up the completion of this flow by utilizing multipath. Determining the maximum number of flows each host can send ($MaxNum$) is left for future work, which is discussed in § 6.4.

5 Evaluation

In this section, we use NS-3 simulations to evaluate the performance of Courier. We compare Courier with the state-of-the-art approaches.

(i) *Hadoop*: MapReduce is executed in the latest Hadoop manner without any coflow scheduling algorithm being deployed. Particularly, two Hadoop mechanisms that notably affect MapReduce performance are implemented in our simulation: (a) the *mapper output combiner*, which automatically aggregates a job's mapper outputs on a server for network transmission, and (b) *reducer flow number restriction*, which limits the number of flows a reducer can initiate [2].

(ii) *Courier*: Courier is deployed on *Hadoop* to merge multiple flows to a unified flow between each pair of hosts. Note that the Hadoop mechanism is still in effect in this scenario.

(iii) *Aalo + Hadoop*: Aalo is deployed on *Hadoop*, where coflows are scheduled in a non-clairvoyant manner, and the number of concurrent flows is restricted in Hadoop's manner.

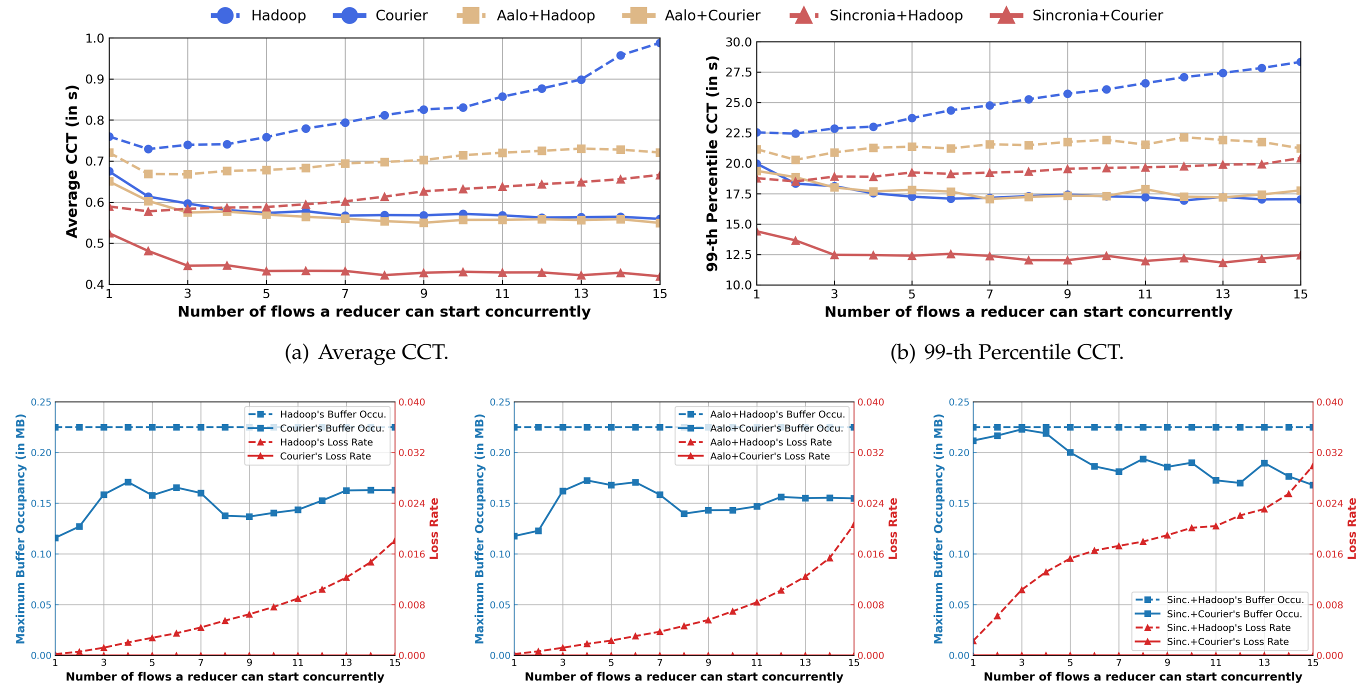
(iv) *Aalo + Courier*: Aalo is deployed on *Courier*, where multiple flows are transmitted as one flow while maintaining the scheduling order of Aalo.

(v) *Sincronia + Hadoop*: Sincronia is deployed on *Hadoop*, in which the flow order enforcement and rate allocation are offloaded to the DSCP mechanism, and the number of concurrent flows is restricted in Hadoop's manner.

(vi) *Sincronia + Courier*: Sincronia and Courier is deployed, where multiple flows between each pair of servers are merged into a unified flow, whose DSCP is the highest DSCP of all flows being merged.

Topology: In the simulation experiments, we adopt a simple big switch topology, which is widely used in the evaluation of coflow scheduling algorithms [7], [8], [10], [13], [15], [39]. In experiments of § 5.1 and § 5.3, 16 servers are connected to the big switch by 1Gbps, 1µs links, which is similar to [9], [39]. The buffer size of the big switch is 0.225 MB (about 150 MTU) per port. The large switch topology facilitates us to study the relationship between the number of concurrent flows and the queue length. Therefore, we

2. For more details on the two mechanisms, please refer to *Shuffle.java* [37] and *Fetcher.java* [38] in Hadoop 3.4.



(a) Average CCT.

(b) 99-th Percentile CCT.

(c) Buffer Occupancy & Loss Rate of *Hadoop* and *Courier*.

(d) Buffer Occupancy & Loss Rate of *Aalo* w/ and w/o Courier.

(e) Buffer Occupancy & Loss Rate of *Sincronia* w/ and w/o Courier.

Fig. 4. Evaluation of w/ and w/o Courier under different numbers of concurrent flows per reducer.

can clearly reveal the benefits brought by Courier. In experiments of § 5.2.1, a fat tree topology with $k = 6$ is used to verify the multipath utilization of Courier. In experiments of § 5.2.2, a big switch topology with 512 servers was used to verify Courier's ability to mitigate the concurrent flow problem in the large topology. The link and buffer settings in § 5.2 are the same as those in the previous subsections.

Workload: Unless stated otherwise, our workload is based on a MapReduce trace collected from a 3000-machine, 150-rack Facebook cluster [7]. The coflows in the trace are scaled to match the bandwidth of our topology. In § 5.3, we changed the coflow size (from *small* to *big*) and inter-arrival time (from *infrequent* to *frequent*) in the same way as the evaluation of Sincronia [10], generating four different loads: small and infrequent(S-I), small and frequent (S-F), big and infrequent(B-I), big and frequent (B-F).

Parameters: DCTCP is deployed as the transport layer protocol in all experiments. In order to avoid unsuitable RTO causing CCT to rise too much in packet loss scenarios, we choose 20 µs as a suitable RTO based on experimental results (Fig. 3). The ECN (explicit congestion notification) marking threshold is set to 30KB (about 20 MTU), and the estimation weight g is set to $1/16$ as recommended in [19]. The parameters of scheduling algorithms all use the recommended values in [8], [10], [39]. Unless stated otherwise, we configure the number of reducers per server to 20 and the number of concurrent flows per reducer to 5 (the default value in Hadoop).

Metrics: We use two primary performance metrics: (i) Average and 99-th percentile CCT, the most basic coflow performance metric. (ii) Maximum queue length and packet loss rate, which are key optimization metrics of Courier.

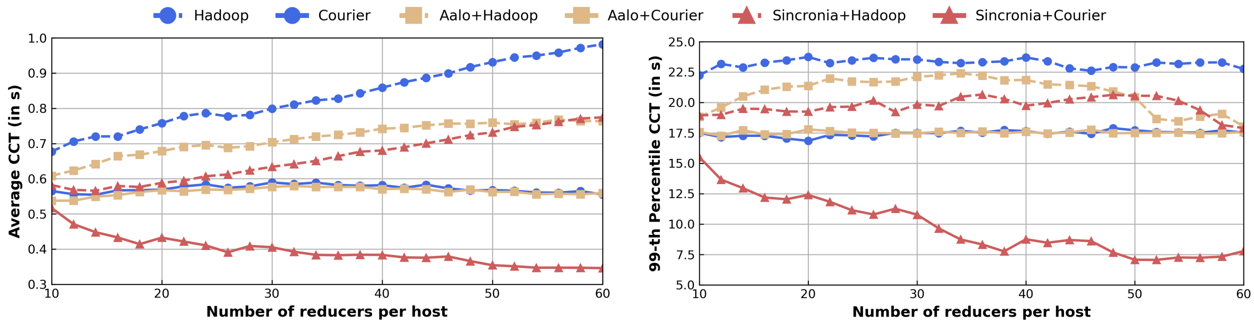
5.1 Number of Concurrent Flows

In this subsection, we evaluate Courier's ability to reduce the queue length and avoid packet loss. As mentioned earlier, there are two main factors that affect the number of concurrent flows in the network: the number of reducers per server (§ 2.3) and the number of concurrent flows per reducer (§ 2.4). We then conduct experiments on each of these two factors separately.

5.1.1 Number of Concurrent Flows per Reducer

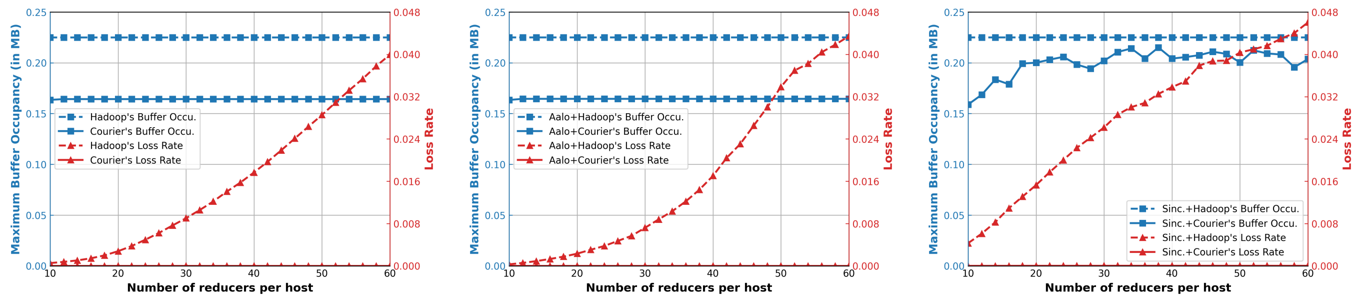
In § 2.4, we discussed the drawbacks of Hadoop's workaround solution. Next, we illustrate the drawbacks of the Hadoop approach and how Courier solves it with the experiments shown in Fig. 4. In the experiment, the number of reducers per server is set to 20, and the number of concurrent flows that can be started per reducer changes from 1 to 15 in step of 1.

Fig. 4(a), 4(b), and 4(c) show the evaluation results of *Hadoop* and *Courier* under the different numbers of flows per reducer. The difference in average CCT between *Hadoop* and *Courier* is insignificant when the number of flows per reducer is relatively small (about 1-5). This is because packet loss has not yet become the dominant factor of *Hadoop*'s coflow performance at this point. As the number of flows per reducer increases, the CCT of *Hadoop* continues to increase due to gradually severe packet loss, while the CCT does not increase significantly in *Courier*. The evaluation of the 99-th percentile CCT shown in Fig. 4(b) is similar to the average CCT, indicating that Courier does not cause long tails. On average, the *Courier*'s average CCT is 22% lower than the *Hadoop*'s and the 99th CCT is 23% lower. Fig. 4(c) reveals the reason why Courier can shorten CCT. Without Courier, *Hadoop*'s maximum buffer occupancy always



(a) Average CCT.

(b) 99-th Percentile CCT.



(c) Buffer Occupancy & Loss Rate of *Hadoop* and *Courier*.

(d) Buffer Occupancy & Loss Rate of *Aalo* w/ and w/o Courier.

(e) Buffer Occupancy & Loss Rate of *Sincronia* w/ and w/o Courier.

Fig. 5. Evaluation of w/ and w/o Courier under different numbers of reducers per server.

reaches the buffer limit of the switch. As a result, in *Hadoop*, the packet loss rate rises continuously with the number of flows, which leads to an increase in CCT. In contrast, *Courier* can effectively reduce the buffer occupancy. Thus packet loss is avoided, and CCT is not affected.

Fig. 4(a), 4(b), and 4(d) shows the evaluation of *Aalo + Hadoop* and *Aalo + Courier*. When the number of flows that each reducer can start is small (about 1-2), Aalo's scheduling ability is impaired by the limitation of the number of flows. The impairment of scheduling ability is reflected in the CCT rise in both *Aalo + Hadoop* and *Aalo + Courier*. As the number of flows increases, Aalo's scheduling ability gradually recovers, but *Aalo + Hadoop*'s CCT gradually rises because packet loss affects the performance. In contrast, the CCT of *Aalo + Courier*, where no packet loss occurs, gradually decreases as the number of flows increases, reflecting the improvement in scheduling capability. The experimental results of the average CCT and 99-th percentile CCT (Fig. 4(b)) are similar. The 99-th percentile CCT is more volatile because it is dominated by a few large coflows. Thus it mainly depends on how many times the large coflows have experienced packet loss and retransmissions, which is more accidental. *Aalo + Courier* consistently has a lower CCT than *Aalo + Hadoop* (and *Courier*), proving that Courier can be well integrated with Aalo.

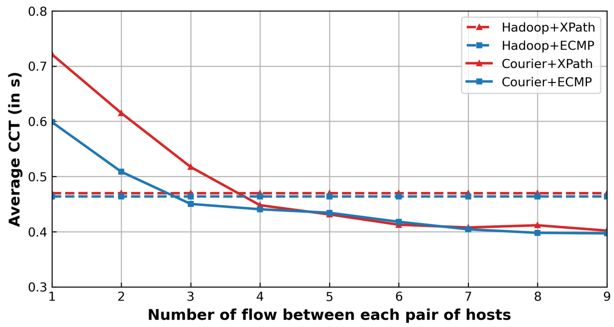
The performance of *Sincronia + Hadoop* and *Sincronia + Courier* shown in Fig. 4(a), 4(b), and 4(e) is similar to that of *Aalo + Hadoop* and *Aalo + Courier* illustrated above.

Comparing all six scenarios, we can derive three conclusions. (i) On average, the loss rates of *Aalo + Hadoop* and *Hadoop* are basically the same, and the loss rate of *Sincronia + Hadoop* is 167% of *Hadoop*'s. This is because the DSCP-enabled mechanism used by Sincronia actually reduces the available buffer space for each priority. Thus, while the buffer limit is still not reached, *Sincronia + Courier*'s buffer occupancy is higher than *Courier* and *Aalo + Courier*. (ii) It can be found that the CCT of *Sincronia + Hadoop* (*Sincronia + Courier*) is better when compared with that of *Aalo + Hadoop* (*Sincronia + Courier*). This is due to the clairvoyant nature of Sincronia makes it has a stronger scheduling capability than Aalo. Additionally, we can clearly see that the CCT of *Sincronia + Courier* is lower than the other five scenarios, and the CCT keeps decreasing as the number of flows increases. This observation shows that the combination of Courier and scheduling algorithms can avoid the negative effect of packet loss, thus fully exploiting the scheduling capability. (iii) In all experiments, Courier effectively controls the queue length, avoids packet loss, and reduces CCT while avoiding the long tail in CCT. This shows that Courier has the ability to control the number of concurrent flows in the network at different numbers of flows per reducer, delivering better network performance.

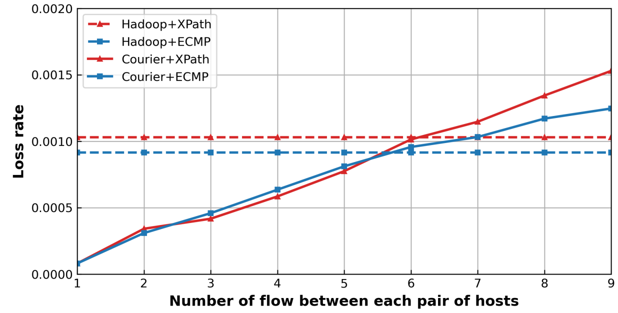
5.1.2 Number of Reducers per Server

As described in § 2.3, the trend of placing an increasing number of cores on a single server will eventually lead to an increase in the number of reducers on a machine, and thus an increase in the number of concurrent flows. We conduct the experiments shown in Fig. 5 to verify whether Courier can reduce the queue length and avoid packet loss as the number of reducers increases. In the experiment, the number of flows that each reducer can start is set to 5, which is the default value in Hadoop, and the number of reducers on each machine changes from 10 to 60 in step of 2.

The results of the experiment on the number of reducers per server are shown in Fig. 5. The performance of this

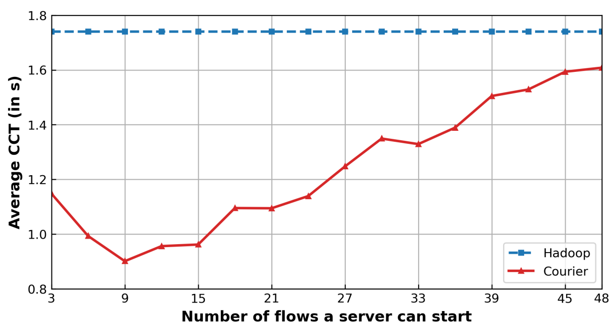


(a) Average CCT.

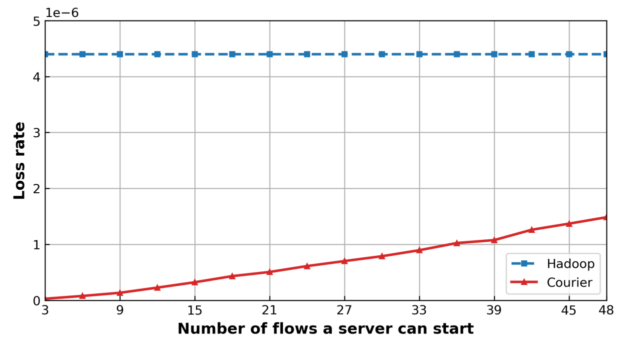


(b) Loss rate.

Fig. 6. Evaluation of Courier under a multipath topology.



(a) Average CCT.



(b) Loss rate.

Fig. 7. Evaluation of Courier under a topology with 512 servers.

experiment is similar to the previous one since the primary influence on the network performance is the number of concurrent flows in both experiments. This experiment has a main difference compared to the previous experiment. It can be observed that the CCT of *Courier* increases as the number of reducers per server increases. This is because Courier divides the uniform flow rate evenly among each origin flow when there is no scheduling algorithm, which in fact causes blocking to small coflows when the number of reducers on each server increases. The blocking is mitigated or disappears in *Aalo + Courier* and *Sincronia + Courier*.

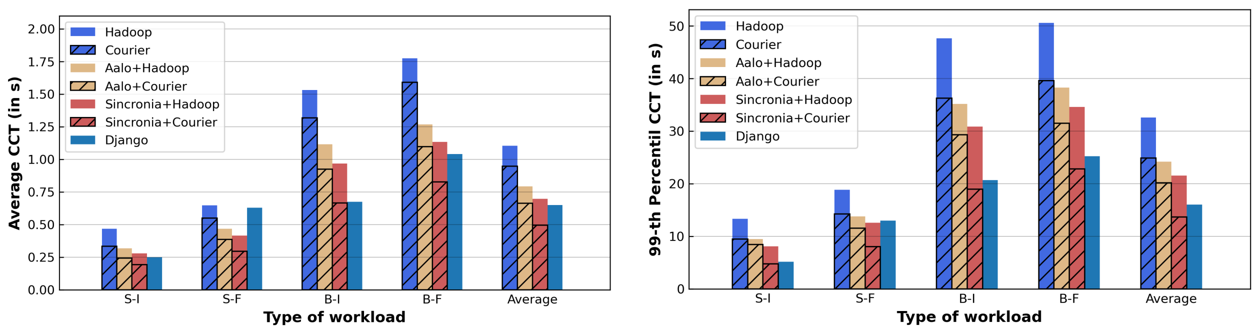
When the number of reducers per server exceeds 50, the 99-th percentile CCT decreases for the three methods not utilizing Courier. This is because, in our 526 coflows workload, the 99th percentile CCT is determined by the largest five coflows. With an increased number of reducers per server, these large coflows can more quickly utilize the additional reducers for initiation. This phenomenon does not contradict our conclusion that Courier can reduce queue lengths and packet loss rates, thereby lowering the CCT, as a significant reduction in both average CCT and 99th percentile CCT can be observed after implementing Courier.

5.2 Courier in Various Topologies

A flexible flow number control mechanism is designed to make Courier compatible with multipath topology (§ 4.2.2) and large topology (§ 4.2.1). In this subsection, we evaluate the performance of Courier in both topologies separately. To reveal the gains of Courier's flow number control mechanism, no scheduling algorithm is deployed.

5.2.1 *Multipath Topology*

Fig. 6 shows the results of the simulation experiments in a fat-tree topology with $k = 6$. We compare the performance of *Hadoop* with *Courier* under different numbers of flows between each pair of servers. In this experiment, two multipath routing methods, ECMP and XPath, were deployed separately. As the horizontal coordinate represents the number of flows in *Courier*, the CCT and loss rate of *Hadoop* are shown as a horizontal line. Note that we have only implemented the path enforcement mechanism of XPath, but not the path selection mechanism, which is beyond this paper's scope. Therefore, *Courier + XPath* suffers from poor utilization of the multipath topology when the number of flows per pair of servers is small. When the number of flows is greater than 4, there is almost no difference between the performance of XPath and that of ECMP. It is observed the CCT of *Courier* is lower than that of *Hadoop* when the number of flows increases to 4 or more, regardless of which multipath method is used. When the number of flows between each pair of servers is lower than 6, the packet loss rate of *Courier* is lower than the packet loss rate of *Hadoop*. When the number of flows is higher than 6, *Courier*'s packet loss rate is higher than *Hadoop*'s because *Courier* artificially utilizes more paths (starts more flows than *Hadoop*). However, even with a higher packet loss rate,



(a) Average CCT. (b) 99-th Percentile CCT.

Fig. 8. Evaluation of Django and other algorithm w/ Courier.

Courier's CCT is lower than *Hadoop*'s because Courier does utilize more paths.

5.2.2 Large Topology

Figure 7 shows our evaluation of Courier in a large topology with 512 servers. Identical to the previous experiment, the horizontal coordinate represents the number of flows a *server* can start in *Courier*. Thus the CCT and loss rate of *Hadoop*, where the number of flows a *reducer* can start is set to 5 (the default value in Hadoop), are shown as a horizontal line. When the number of flows that can be started per server is too small, *Courier*'s average CCT increases because of head-of-line blocking. And when there are too many flows that can be started per server, the average CCT of *Courier* increases because of the increase in packet loss. When the number of flows per server is between 5 and 15, *Courier* effectively balances the two influencing factors of head-of-line blocking and packet loss and reduces the average CCT. In contrast, *Hadoop*'s average CCT and packet loss rates are inferior to *Courier*'s because it cannot validly limit the number of flows.

5.3 Comparison with Django

Django is a rate-based clairvoyant scheduling algorithm that solves the concurrent flow problem by limiting the number of flows. In this subsection, we compare *Django* with the six scenarios under the four workloads mentioned before. In Fig. 8, we use four colored bars to show the CCT of *Hadoop*, *Aalo + Hadoop*, *Sincronia + Hadoop*, and *Django*. Then, we use three colored shaded bars to show the CCT of *Courier*, *Aalo + Courier*, and *Sincronia + Courier*. To facilitate the comparison of the CCT with or without Courier, we plot the corresponding experiment results together. It should be noted that we do not implement Django's machine learning model. Instead, we mimic the machine learning training process by running the simulator multiple times with different numbers of flows and directly selecting the best number of flows. No packet loss occurs in the experiments with Courier and Django.

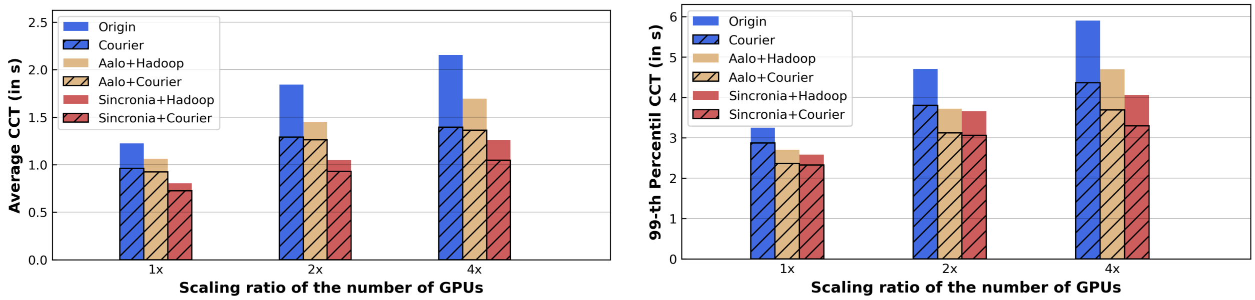
In Fig. 8 it can be observed that *Hadoop* and *Courier* have the longest average CCT because no scheduling algorithm is used. *Aalo*'s CCT is higher than *Sincronia*'s because the latter is a clairvoyant scheduling algorithm. On average over all workloads, Courier reduces the average CCT and 99-th percentile CCT for both Hadoop by about 14%, reduces the average CCT for Aalo by about 16% and 99-th percentile CCT by about 17%, reduces the average CCT for Sincronia by about 29% and 99-th percentile CCT by about 37%. It can be observed that Courier can reduce the average CCT and 99-th percentile CCT at all workloads.

Looking at the average CCT of *Django* (Fig. 8(a)), it can be found that *Django* performs the worst under the *S-F* workload (almost the same as *Hadoop*). This is because Django's flow number limit mechanism blocks small coflows. The smaller the size of the coflows, the more significant the impact of blocking, and the more frequently the coflows launch, the greater the chance of blocking. This also explains why *Django* performs best under *B-I* workloads. On average over all workloads, *Django*'s average CCT is lower than *Aalo + Hadoop*'s but basically comparable to *Aalo + Courier*'s. However, comparing *Django* with *Sincronia* on average over all workloads with Django, we can observe that *Sincronia + Hadoop*'s CCT is 8% higher than *Django*. But with the help of Courier, *Sincronia + Courier* outperform *Django* by 34% on average over all workloads. The 99-th percentile CCT (shown in Fig. 8(b)) shows a similar trend to the average CCT, where *Django* performs worst on *S-F* workload, performs best on *B-I* workload and performs inferior to *Sincronia + Courier* over all workloads.

In this section, we demonstrate the following points. (i) Django successfully avoids packet loss by limiting the number of flows and thus obtains a lower CCT than normal scheduling algorithms. (ii) But Django's scheduling-based flow number control mechanism inherently limits its scheduling capability. By combining with Courier , mainstream algorithms can achieve better results than Django.

5.4 Courier for Distributed DNN Training

In this subsection, we validate the effectiveness of Courier in distributed DNN training, a cluster computing scenario that has been significantly important recently. We utilize Microsoft's distributed DNN training trace [4], which employs data parallelism and parameter server (PS) in the training. In each training iteration, GPUs first send gradients to the PSs, which aggregate these gradients and send the results back, constituting two coflows. We adopt the method from [40] for coflow generation. The coflow size matches the model size provided by [4]. For each coflow, we construct



(a) Average CCT.

(b) 99-th Percentile CCT.

Fig. 9. Evaluation of w/ and w/o Courier in distributed DNN training with different number of GPUs.

PSs equal to half the number of GPUs. Considering the trend of increasingly larger models requiring more GPUs, we scale the number of GPUs in the traces by 2× and 4× to evaluate the performance improvements of Courier under this trend. Our experiments were conducted on 16 servers, each equipped with 8 GPUs, representing the current typical GPU configuration for distributed DNN training scenarios [41].

The experimental results are presented in Figure 9. We compared the performance of Origin (no restriction or optimization), Aalo, Sincronia, and the performance of these three methods with Courier. The results show that Courier significantly reduces CCT, and this reduction increases with the number of GPUs. When the scale ratio is 1×, 2×, and 4×, the average CCT improvement of Courier over Origin is 12%, 19%, and 26%, respectively, while the 99-th percentile CCT improvement is 21%, 30%, and 35%. On Aalo and Sincronia, Courier exhibits a similar trend of improvement. In the distributed DNN training scenario, the impact of the number of GPUs on CCT is caused by the concurrent flow issue, which is shown to be effectively addressed by Courier.

Insights: These experiment results make four key conclusions. First, too many concurrent flows will lead to packet loss, thus affecting CCT. Courier can effectively reduce the number of concurrent flows, thus mitigating packet loss and reducing CCT in various scenarios and topologies. Second, Courier integrates well with mainstream scheduling algorithms, bringing a 16% to 29% CCT acceleration ratio on average over all workloads. Third, the combination of Courier and mainstream scheduling algorithms outperforms the state-of-the-art scheduling-based flow number control method (Django) by about 30%. Finally, Courier brings significant performance improvements on average CCT in both traditional (29%) and emerging (26%) cluster computing jobs.

6 Discussion

6.1 Other Congestion Control Protocols

There are a variety of congestion control protocols that can be used in the data center, but we mainly concentrate on DCTCP, which is one of the most popular protocols. In this subsection, we will discuss Courier's benefits on other congestion control protocols. Existing congestion control protocols can be classified into two categories, that is window-based and rate-based.

6.1.1 *Window-based congestion control protocol*

HPCC [42] leverages in-network telemetry (INT) to obtain accurate link load information for precise congestion control. The excessive number of concurrent flows incurs two problems for HPCC. First, HPCC needs to make a tradeoff in the step of additive increase, W_{AI}. A larger W_{AI} can improve the speed of convergence to fairness but can result in poorer tolerance of concurrent flows. In this aspect, Courier can help HPCC sustains a larger W_{AI} by maintaining a stable number of concurrent flows. Second, HPCC's flow scheduler consumes a large number of hardware clocks; thus, the number of concurrent flows is limited by the clock frequency of the hardware. The FPGA used in the HPCC prototype only supports 300 concurrent flows per interface. In this aspect, Courier can be used to reduce HPCC's hardware clock burden.

Another representative window-based congestion control protocol is Swift [43], which uses the end-to-end delay as a congestion signal. It can handle a higher degree of incasts and has at least a 10x lower loss rate than DCTCP. However, Swift's average queue length grows as $O(\sqrt{N})$ under the assumption that there are N concurrent flows that equally share the bandwidth and have random start times passing through a link. Thus, although the problem is not as severe as in DCTCP, concurrent flow problem still exists in Swift. With Courier, Swift can accommodate more concurrent flows.

6.1.2 *Rate-based congestion control protocol.*

There are two characteristics of the majority of rate-based congestion control protocols (*e.g.* DCQCN [44], TIMELY [45]). First, in order to fully utilize the bandwidth during the first RTT and thus accelerate the completion of small flows, flows start at a high rate or line-rate with rate-based congestion controls. This behavior puts a lot of pressure on the switch's buffers. Using Courier can mitigate this negative impact by reducing the frequency of starting new flows. Second, in case the congestion signal is delayed due to congestion, the congestion controls use a window to limit the volume of outstanding data. A small window can lead to unnecessary throughput loss during network fluctuations, while a large window can further worsen the situation during congestion. Assuming that there are N congested flows with a window of W, the total amount of data in the network during congestion is $N \times W$. Courier keeps the number of concurrent flows N stable, thus reducing network load $N \times W$ during congestion, and increases the choice space for the window W.

6.2 Other Coflow Scheduling Algorithms

The paper mainly focuses on scheduling algorithms similar to Aalo and Sincronia. In addition, there are also many scheduling algorithms that use different scheduling mechanisms.

RAPIER [9] is a scheduling algorithm that integrates routing and scheduling for better performance. It is closer to the reality of data center networks than other scheduling algorithms that abstract data center networks as a large switch, but it also does not take into account the stress that concurrent flows put on the switches. RAPIER specifies both path and rate for each flow, so Courier can still aggregate flows on the same path as the per-flow rate allocation scheduling algorithm (§ 4.1).

There are also many algorithms focused on theoretical analysis [13], [14], [15], [16], [17], [18]. However, they always model the data center network as a graph and ignore the realistic details of switches. Most theoretical scheduling algorithms use the per-flow rate allocation mechanism as the scheduling assurance mechanism, so Courier can be integrated directly with these algorithms.

6.3 Implementation

Courier can be fully implemented on the host without requiring any changes within the network (such as switches). In the following discussion, we explore two methods for implementing Courier. (i) Modifications to the network protocol stack. Courier needs to merge multiple flows issued by the application (*i.e.*, computing framework) into one network flow (§ 4.2.1) or split a single application flow into multiple network flows (§ 4.2.2). This can be implemented by decoupling the application flows and the network flows within the network protocol stack. An existing example is Snap [46], where the network stack is divided into two sub-layers. The upper layer provides traditional APIs to applications, while the lower layer manages network flows and maintains a *flow mapper* that maps application layer flows to network flows. Courier can be implemented by modifying the *flow mapper* in Snap. (ii) Modifications to the cluster computing framework. In scenarios such as cloud computing where users cannot modify the network protocol stack, Courier can be implemented by modifying the cluster computing framework. For instance, Hadoop offers *Pluggable Shuffle* interface that allows customization of the data transmission scheme during the shuffle process.

Note that Courier is transparent to the upper-layer computing frameworks and does not alter the input or output of the computation. Take Hadoop as an example. In Hadoop, the mapper first writes the data to the disk. During the shuffle process, the reducer transfers the data in 64 KB blocks to the local disk, and computation begins once the transfer is complete. Courier also transfers the data of each individual flow in 64 KB blocks after flows are merged. These blocks are written to their designated disk regions as determined by Hadoop. Upon completion of all transfers, Courier notifies Hadoop's reducer to begin computation. During the transmission process, each reducer receives identical data as in the absence of Courier. Thus, the outcome remains unchanged.

6.4 Determine the Optimal Flow Number

To the best of our knowledge, no existing work has completely resolved the issue of determining the optimal number of flows within a network. We plan to leave this issue as the future work and employ a method similar to Django that uses machine learning models to predict the optimal flow number. Different from Django which struggles to predict the optimal flow number precisely, we found that when using Courier, the CCT performance remains superior across a range of concurrent flow numbers, allowing some margin of error in predictions. As shown in Fig. 6(a), in a multipath topology, the CCTs with Courier are nearly identical and significantly better than without Courier when flows per host pair range from 6 to 9. Fig. 7(a) demonstrates that in a large topology, the CCTs are reduced by 51% to 57% with Courier for 6 to 15 flows per host. Therefore, predicting the optimal number of flows for Courier is simpler, as some prediction errors result in only a graceful performance degradation. We believe it is worthwhile to explore in the future work.

7 Conclusion

In this paper, we propose a unified communication agent in cluster computing, called Courier. Our work is mainly motivated by the concurrent flow problem in cluster computing frameworks and the drawbacks of existing solutions. By deploying Courier in data center, the number of cluster computing flows between any pair of servers is reduced to 1. To make Courier practical, we have designed it carefully so that it can be combined with existing scheduling algorithms and perform well under multipath topologies and large topologies. We conducted extensive experiments to prove that Courier can effectively solve concurrent flow problems, integrates well with existing scheduling algorithms, and outperforms the state-of-the-art approach by about 30%. In this paper, we only study the performance of Courier under DCTCP. However, we believe that Courier has the ability to perform well under other mainstream congestion control protocols.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. This research is supported by the Key Program of Natural Science Foundation of Jiangsu under grant No. BK20243053, the National Natural Science Foundation of China under Grant Numbers 62325205, 62072228, and 62172204 , and the Future Network Scientific Research Fund Project FNSRFP-2021-ZD-02.

REFERENCES

[1] Zhaohua Wang, Zhenyu Li, Guangming Liu, Yunfei Chen, Qinghua Wu, and Gang Cheng. Examination of wan traffic characteristics in a large-scale data center network. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 1–14, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[3] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 98–109, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[5] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[6] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, page 31–36, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 443–454, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[9] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 424–432. IEEE, 2015.

[10] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 16–29, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 1–15, New York, NY, USA, 2024. Association for Computing Machinery.

[12] Apache Software Foundation. Apache hadoop. https://hadoop.apache.org/, 2024.

[13] Zhen Qiu, Cliff Stein, and Yuan Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 294–303, 2015.

[14] Samir Khuller and Manish Purohit. Brief announcement: Improved approximation algorithms for scheduling co-flows. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 239–240, 2016.

[15] Mehrnoosh Shafiee and Javad Ghaderi. An improved bound for minimizing the total weighted completion time of coflows in datacenters. *IEEE/ACM Transactions on Networking*, 26(4):1674–1687, 2018.

[16] Saba Ahmadi, Samir Khuller, Manish Purohit, and Sheng Yang. On scheduling coflows. *Algorithmica*, 82(12):3604–3629, 2020.

[17] Hamidreza Jahanjou, Erez Kantor, and Rajmohan Rajaraman. Asymptotically optimal approximation algorithms for coflow scheduling. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 45–54, 2017.

[18] Mosharaf Chowdhury, Samir Khuller, Manish Purohit, Sheng Yang, and Jie You. Near optimal coflow scheduling in networks. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 123–134, 2019.

[19] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[20] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):73–84, 2011.

[21] Nandita Dukkipati, Neal Cardwell, Yuchung Cheng, and Matt Mathis. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01, Internet Engineering Task Force, February 2013. Work in Progress.

[22] Claudio DeSanti. Ieee 802.1: 802.1 qbb-priority-based flow control, 2009.

[23] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 17–28, 2014.

[24] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. Fastlane: making short flows shorter with agile drop notification. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 84–96, 2015.

[25] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[26] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. *ACM SIGCOMM computer communication review*, 39(4):303–314, 2009.

[27] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 422–434, 2020.

[28] Kexin Liu, Chen Tian, Qingyue Wang, Hao Zheng, Peiwen Yu, Wenhao Sun, Yonghui Xu, Ke Meng, Lei Han, Jie Fu, et al. Floodgate: taming incast in datacenter networks. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 30–44, 2021.

[29] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. Backpressure flow control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, 2022.

[30] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[31] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 63–74, 2009.

[32] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

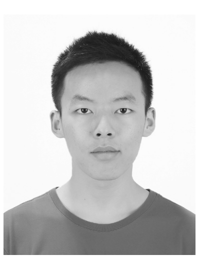
[33] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[34] Cristian Hetnandez Benet, Andreas Kassler, Gianni Antichi, Theophilus A Benson, and Gergely Pongracz. Providing in-network support to coflow scheduling. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 235–243. IEEE, 2021.

[35] Christian Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, November 2000.

[36] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 15–28, 2015.

[37] Apache Hadoop. Default parameters of mapreduce in hadoop 3.4.1. https://github.com/apache/hadoop/blob/branch-3.4/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/task/reduce/Shuffle.java, 2022.
[38] Apache Hadoop. Fetcher.java in hadoop 3.4. https://github.com/apache/hadoop/blob/branch-3.3/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/task/reduce/Fetcher.java, 2024.
[39] Jiaqi Zheng, Liulan Qin, Kexin Liu, Bingchuan Tian, Chen Tian, Bo Li, and Guihai Chen. Django: Bilateral coflow scheduling with predictive concurrent connections. *Journal of Parallel and Distributed Computing*, 152:45–56, 2021.
[40] Wenxin Li, Sheng Chen, Keqiu Li, Heng Qi, Renhai Xu, and Song Zhang. Efficient online scheduling for coflow-aware machine learning clusters. *IEEE Transactions on Cloud Computing*, 10(4):2564–2579, 2020.
[41] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training llms with trillion parameters, 2024.
[42] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.
[43] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020.
[44] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.
[45] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.
[46] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

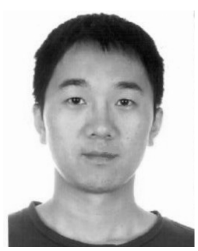


Zhaochen Zhang received the B.S. degree at the Department of Software Engineering, Huazhong University of Science and Technology, China, in 2021. He is working towards the Ph.D. degree in the Department of Computer Science and Technology at Nanjing University, China. His research interests include datacenter networks.

Xu Zhang received the BS degree in communication engineering from Beijing University of Posts and Telecommunications, China, in 2012 and the Ph.D. degree in computer science from the Department of Computer Science and Technology, Tsinghua University, China, in 2017. His research interests include AI for computing and networking, multimedia networks, Internet of Things and Metaverse. He is a recipient of the EU Marie Sklodowska-Curie Individual Fellowships and a co-recipient of 2019 IEEE Broadcast Technology Society Best Paper Award.

Zhaoxiang Bao is a 3rd-year undergraduate and pursuing his bachelor degree at Nanjing University. Currently he works as an intern at State Key Laboratory for Novel Software Technology, Nanjing University. His research interests include data center networks.

Liang Wei is currently the team director of Jiangsu future network innovation research institute , and the leader of SDN network group of national major science and technology infrastructure future network test facility project. Lead the team in the research and development of future network architecture and cloud network integration system. The main research fields include: software defined network, deterministic network, cloud computing, network test bed, etc.



Chaohong Tan received the master's degree in computer application technology from Jiangnan University in 2011. He is currently the technical manager of Jiangsu Future Networks Innovation Institute. His research interests include software defined network, cloud network integration and network function virtualization.

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.

Chen Tian is a 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, distributed systems, Internet streaming and urban computing.