Optimizing Network Transfers for Data Analytic Jobs Across Geo-Distributed Datacenters

Li Chen, *Member, IEEE*, Shuhao Liu, and Baochun Li, *Fellow, IEEE*

Abstract—It has become a recent trend that large volumes of data are generated, stored, and processed across geographically distributed datacenters. When popular data parallel frameworks, such as MapReduce and Spark, are employed to process such geo-distributed data, optimizing the network transfer in communication stages becomes increasingly crucial to application performance, as the inter-datacenter links have much lower bandwidth than intra-datacenter links. In this article, we focus on exploiting the flexibility of multi-path routing for inter-datacenter flows of data analytic jobs, with the hope of better utilizing inter-datacenter links and thus improve job performance. We design an optimal multi-path routing and scheduling strategy to achieve the best possible network performance for all concurrent jobs, based on our formulation of an optimization problem that can be transformed into an equivalent linear programming (LP) problem to be efficiently solved. As a highlight of this article, we have implemented our proposed algorithm in the controller of an application-layer software-defined inter-datacenter overlay testbed, designed to provide transfer optimization service for Spark jobs. With extensive evaluations of our real-world implementation on Google Cloud, we have shown convincing evidence that our optimal multi-path routing and scheduling strategies have achieved significant improvements in terms of job performance.

Index Terms—Geo-distributed datacenters, network transfer, data analytics, optimization

1 Introduction

It becomes increasingly typical for global services that large volumes of data are generated all over the world and are stored in their nearest datacenters, which are geographically distributed. For example, services deployed by Microsoft and Google [1], [2] routinely span multiple geo-distributed datacenters and generate large volumes of data for analysis, such as user activity and system monitoring logs. To efficiently process data at such a large scale, popular data parallel frameworks, such as MapReduce [3] and Spark [4], are employed extensively. They proceed in multiple stages, each consisting of a number of tasks in parallel. Between consecutive stages, intermediate data are fetched from the preceding stage, which generates a number of network flows.

Traditionally, a data parallel job runs within a single datacenter, where all its input data are stored. Network flows generated during data processing are bounded within the intra-datacenter network. However, when input data are distributed across multiple datacenters, it is inevitable that flows across datacenters will be generated to traverse inter-datacenter links. As bandwidth on inter-datacenter network links is limited [5], optimizing the inter-datacenter network transfer becomes critical towards accelerating such geo-distributed jobs and thus improving their performance.

Existing efforts on improving the performance of geo-distributed data analytic jobs fall into two categories. The first focuses on reducing the total amount of cross-datacenter traffic [5], [6], [7]. However, this does not necessarily reduce the job completion times, since the network transfer time is not only influenced by the traffic size, but also the bandwidth of inter-datacenter links along which the flows traverse. The second category (e.g., [8], [9]) focuses on designing better task placement strategies, based on the fact that different assignment of tasks to datacenters lead to different flow patterns across datacenters, and ultimately, different job completion times.

Yet, all existing works above seek to alleviate the performance degradation of inter-datacenter transfers by modifying the generated traffic pattern across datacenters, which do not directly solve the problem from the perspective of optimizing the network transfers given certain traffic patterns. To fill this gap, we propose to optimize the inter-datacenter transfers by exploiting the path flexibility to better utilize the inter-datacenter link bandwidth, which provides a complementary and orthogonal way to improve the performance of geo-distributed data analytic jobs.

To have a better intuition of the routing flexibility, we present five datacenters of Amazon EC2 in Fig. 1, which are geographically distributed across the world. The thickness of the line between each pair of datacenters is proportional to the amount of available bandwidth. It is clearly shown that the bandwidth of the cross-ocean link between N. Virginia and Singapore is much smaller than any other link. For flows between N. Virginia and Singapore, apart from using the direct path through the narrow link, as illustrated by the solid arrow, we can take the detoured path by first

- *Li Chen is with the School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA 70504 USA. E-mail: li.chen@louisiana.edu.*
- *Shuhao Liu and Baochun Li are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto ON M5S, Canada. E-mail: {shuhao, bli}@ece.toronto.edu.*

Manuscript received 12 Sept. 2020; revised 7 May 2021; accepted 23 June 2021. Date of publication 29 June 2021; date of current version 23 July 2021. (Corresponding author: Li Chen.) Recommended for acceptance by A. Orgerie. Digital Object Identifier no. 10.1109/TPDS.2021.3093232

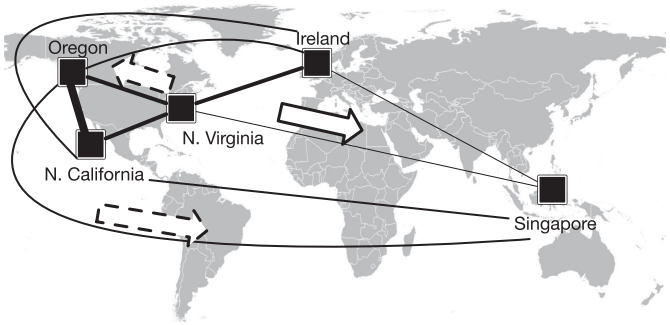


Fig. 1. Flexibility in routing (direct and detoured paths) for inter-datacenter traffic.

forwarding the traffic to Oregon, which then relays the traffic to Singapore, as illustrated by the dashed arrows in Fig. 1. Both links along the detoured path have larger amounts of bandwidth than the direct link. Other detoured paths, such as the one through N. California, can also be considered to accelerate the transfer.

In this paper, we attempt to make the best utilization of the inter-datacenter links by leveraging such flexibility of multi-path routing. Different from existing efforts [10], [11] on multi-path routing, which require that a flow can only take one selected path, we allow each flow to be split and routed through multiple paths, so that the bandwidth can be better utilized to improve the job performance. In particular, given a set of inter-datacenter flows generated by data analytic jobs, we coordinate them with respect to their routing paths and sending rates, so as to accelerate the network transfers and eventually achieve the best possible performance for all the jobs. Specifically, we formulate our problem of multi-path routing and rate assignment as a linear programming (LP), which can be efficiently solved by standard LP solvers (e.g., Mosek [12]) and practically implemented in the real world.

As a highlight of this paper, we have designed and implemented our multi-path routing and scheduling strategy as a transfer optimization service for geo-distributed data analytic jobs. With a special focus on Spark [4], a popular data parallel framework, we have provided a simple and convenient API for it to delegate its traffic. Our transfer service constructs an overlay inter-datacenter network, which follows the principle of Software Defined Networking (SDN) [13] at the application layer. All the inter-datacenter traffic delegated to our service would be fully controlled, following the optimal solution instructed from a centralized controller where our strategy is realized.

We evaluate our prototype implementation of such an optimal transfer service with a wide array of real-world experiments, running various Spark jobs over Google Cloud. Compared with the state-of-the-art, our strategy has shown substantially faster shuffle completion time, up to approximately 30 percent. To the best of our knowledge, this paper presents the first design and implementation of a cloud service for network optimization within geo-distributed data analytics.

The remainder of this paper is organized as follows. In Section 2, we demonstrate the intuition of multi-path routing in improving network utilization and reducing network transfer time for data analytic jobs, which motivates our theoretical investigation of the multi-path routing problem, for a single data analytic job (Section 3) and for multiple jobs concurrently sharing the network (Section 4). We elaborate the design and implementation details of our solution in Section 5, and present our experimental results in Section 6. Related work are surveyed in Section 7 and we conclude our paper in Section 8.

2 Background and Motivation

In this section, we first present an overview of the execution of a data analytic job whose input data is stored across geographically distributed datacenters. Then, motivated by the flexibility of available paths for network flows across these datacenters, we will illustrate the intuitive and immediate benefit of utilizing multiple paths in transferring these flows.

Geo-Distributed Data Analytics. Data parallel frameworks, such as MapReduce and Spark, have become the mainstream platform for today's large-scale data analytics. A typical data analytic job using these frameworks proceeds through several computation stages, each consisting of tens or hundreds of parallel computation tasks. Between consecutive stages, network flows will be generated in the communication stage to transfer the intermediate data for further processing. These flows are collectively defined as a *coflow* of the data analytic job [14]. For example, for a MapReduce job, input data is partitioned into a set of splits to be processed in parallel with map computation tasks. These tasks produce intermediate results, which are then shuffled over the datacenter network to be further processed by reduce computation tasks. In the shuffle phase, the entire set of network flows generated from map tasks to reduce tasks is referred to as a coflow. The completion of a coflow is determined by the slowest of its constituent flows, and speeding up an individual flow does not necessarily improve the coflow performance. The awareness of coflows should be incorporated into flow scheduling and rate allocation to better use bandwidth resources towards job-level performance improvement [14].

When input data of data analytic jobs are globally generated and stored across datacenters in different regions, which becomes increasingly typical for global services provided by today's big companies [5], [6], [8], new challenges arise in running analytic jobs to process such geo-distributed data efficiently. Particularly, there are two alternatives for such data processing. A naive approach is to gather all the input data to a single datacenter and run the traditional data analytic job, which involves an extensive amount of data to be transferred across datacenters. A more advanced alternative is to distribute tasks across datacenters without centralizing the input data, which incurs a smaller amount of intermediate data to be exchanged among datacenters.

In contrast with the datacenter network, the inter-datacenter links have much smaller bandwidth and are heterogeneous with respect to both the available bandwidth and the network loads [15], which make the communication stage easily become the performance bottleneck. Since cross-datacenter transfers are inevitable in geo-distributed data analytics, it becomes crucial to optimize the cross-datacenter

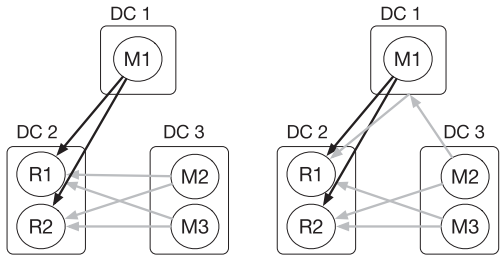


Fig. 2. Routing with direct paths versus Optimal multi-path routing.

network transfers to accelerate job execution and improve job performance.

Flexibility in Path Selection. Inter-datacenter data transfers, in essence, are wide-area network transfers, usually through dedicated links. Google [16] reported that dedicated optical links have been deployed to inter-connect their globally distributed datacenters. As a result, there exist multiple available paths between any pair of datacenters. More importantly, these paths are accessible and controllable by users. For example, to transfer bulk data, one can use a direct transfer between the source and the destination datacenters or, in case of congestion on the direct link, explicitly utilize an intermediate datacenter as a relay to achieve a higher throughput.

In our context of running geo-distributed data analytic jobs, this property can be helpful in terms of reducing the coflow completion times. Fig. 1 shows such a toy example, when a large amount of intermediate data need to be transferred from N. Virginia to Singapore between mappers and reducers. Apparently, the narrow direct link will become the bottleneck, resulting in stragglers and slowing down the completion of the communication stage.

To alleviate the bottleneck, we can route some traffic through alternative paths with larger available bandwidth, such as the detoured path through Oregon or N. California as aforementioned. Intuitively, if all the flows generated in the communication stage can be better balanced across the inter-datacenter links, the slowest flow will finish faster, which will mitigate the adverse impact of stragglers and result in faster completion of the communication stage.

Therefore, our design objective in this paper is to exploit such flexibilities in path selection to optimally coordinate the inter-datacenter network flows of data analytic jobs, so that their communication stages would complete faster and thus the jobs would be accelerated. We would also account for the fairness issue when network flows from multiple jobs are concurrently sharing the inter-datacenter network.

Optimal Multi-Path Routing. We next use a more detailed example to illustrate the basic idea of our multi-path routing in accelerating a data analytic job. As shown in Fig. 2, we consider a MapReduce job with its mapper tasks (`M1`, `M2`, `M3` and others omitted) distributed across 3 datacenters (`DC1`, `DC2`, `DC3`). The reduce tasks `R1` and `R2` are launched in `DC2`, which need to fetch intermediate data from mappers during the shuffle stage. In particular, four flows will be generated from `DC3` to `DC2`, represented by the gray lines, and two flows from `DC1` to `DC2` which are represented by the black lines. (We only illustrate the cross-datacenter flows in the figure.) For simplicity, each of these flows is 100 MB in size, and the link bandwidth between each pair of datacenters is the same (10 MB/s).

TABLE 1
Notations for Transfer Optimization for a Single Job

Notation	Definition
I	Total number of aggregated flows generated by a job
i	Index of the aggregated flow, $i \leq I$
d_i	Traffic volume of the ith aggregated flow
J_i	Total number of available paths for ith aggregated flow
$p(i,j)$	jth path for ith aggregated flow, $j \leq J_i$
$\mathcal{L}$	Set of inter-datacenter links along paths of all aggregated flows in our consideration, $\mathcal{L} = \cup_{i,j} p(i,j)$
l	Inter-datacenter link, $l \in \mathcal{L}$
$\alpha_{i,j}$	Traffic percentage of ith aggregated flow routed through its jth path
b_l	Available bandwidth at link l
t_l	Time to complete all the traffic routed through link l
t	Shuffle completion time, $t = \max_{l \in \mathcal{L}} t_l$

If all these flows are routed through their direct paths as on the left side of Fig. 1, the link between `DC3` and `DC2` would become the bottleneck, resulting in 40 seconds of shuffle completion time (each of the four flows gets a fair share of 2.5 MB/s). However, if we split some traffic from the direct path (`DC3-DC2`) to the alternative path (`DC3-DC1`, `DC1-DC2`), the network load will be better balanced across inter-datacenter links, which naturally speeds up the shuffle phase. Specifically, when 100 MB of traffic is shifted from `DC3-DC2` to the alternative two-hop path, as illustrated by the right side of Fig. 1, both `DC3-DC2` and `DC1-DC2` have the same network load. With this routing, the shuffle completion time is reduced to only 30 seconds (on both `DC3-DC2` and `DC1-DC2` links, each of the three sharing flows gets a fair share of $10/3$ MB/s).

Apart from the flexibility in multi-path routing, when we further consider the inter-datacenter flows from multiple jobs and the flexibility of rate assignment, a larger space can be explored to optimize the network transfers and eventually the job performance for all the sharing jobs, with fairness constraints. Such an extension beyond the simple example would be elaborated with formal formulations and detailed analysis in our later sections.

3 Optimal Transfer for a Single Job

In this section, as a starting point, we will formally construct a mathematical model to study the general problem of multi-path routing for cross-datacenter traffic generated by a single data analytic job, with the objective of minimizing the completion time of the communication stage. Notations are summarized in Table 1.

For a typical MapReduce job, the communication stage is called a *shuffle*, and the set of all the network flows in a shuffle is defined as a *coflow* [14]. In the multi-datacenter scenario, the group of flows in a coflow that have the same pair

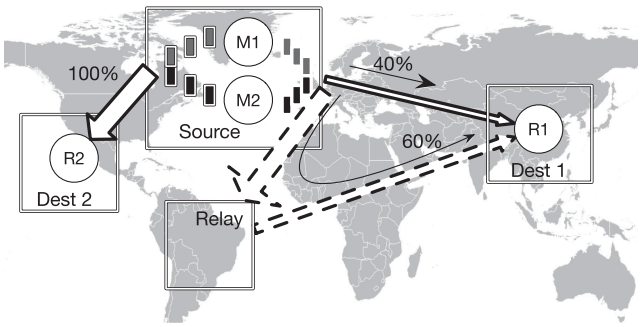


Fig. 3. Illustration of aggregated flow and its multi-path routing.

of source and destination datacenters is called an *aggregated flow*, which is treated as the basic unit in our multi-path routing problem for convenience.

In our consideration, a job generates I aggregated flows in its shuffle. The volume of traffic for the ith aggregated flow is denoted by d_i, which is the total amount of data to be sent by all the consisting flows. Each aggregated flow has the flexibility to take either the direct path, through the link between the source and destination datacenters; or take the detoured path, using other datacenters as relays. The number of total available paths for the ith aggregated flow is denoted as J_i, among which the jth ($j \leq J_i$) path is denoted as $p(i,j)$, the set of inter-datacenter links along the path.

Since different links may have different amounts of traffic, with the flexibility of multi-path routing, the network load can be better balanced across all the links, so that the whole shuffle stage can complete faster. Therefore, we would like to design the optimal multi-path routing algorithm that minimizes the shuffle completion time, i.e., the completion time of the slowest aggregated flow. The decision variables in our routing problem are denoted by $\alpha_{i,j}, \forall i \in \{1,2,\ldots,I\}, \forall j \in \{1,2,\ldots,J_i\}$, representing the percentage of traffic of the ith aggregated flow that is routed through its jth path.

We present an illustrative example of aggregated flows and multi-path routing in Fig. 3. Two mappers ($M1, M2$) at the source datacenter will send intermediate data to two reducers ($R1, R2$) located at two different destination datacenters. The two flows $M1-R2$ and $M2-R1$ have the same pair of source and destination datacenters, and thus form an aggregated flow. This aggregated flow takes two paths, with 40 percent traffic routed along the direct inter-datacenter link and 60 percent detoured through a relay datacenter, as illustrated in the figure. Another aggregated flow consists of the two flows $M1-R2$ and $M2-R2$, which only takes the direct path. The percentages along all the available paths for an aggregated flow are the variables to be determined in our optimization problem.

Let $\mathcal{L} = \cup_{i,j} p(i,j), \forall i, j \in \{1,2,\ldots,J_i\}$ denote the set of all the inter-datacenter links along paths of all the aggregated flows in our consideration. For each link $l \in \mathcal{L}$, the available link bandwidth is represented by b_l. If l is along the jth path of the ith aggregated flow, *i.e.*, $l \in p(i,j)$, α_{ij} percentage of the ith aggregated flow will pass through this link, which increases the link load by the amount of $d_i\alpha_{ij}$. Since l may belong to multiple paths of a particular aggregated flow, and may also belong to paths of different aggregated flows, the total amount of traversing traffic is calculated as the summation of the traffic load over all the possible paths j ($j \in \{1,2,\ldots,J_i\}$) of a particular aggregated flow i it belongs to ($l \in p(i,j)$), and further over all the considered aggregated flows $i \in \{1,2,\ldots,I\}$, which is eventually represented as $\sum_i \sum_{j,l\in p(i,j)} d_i\alpha_{ij}$. Therefore, the time it takes to complete all the traffic routed through link l is calculated as $t_l = \sum_i \sum_{j,l\in p(i,j)} d_i\alpha_{ij}/b_l$.

Since the shuffle completion time is the slowest completion time among all the aggregated flows, and the completion time of an aggregated flow is determined by the bottleneck link along all the paths, we can easily represent the shuffle completion time as $t = \max_{l\in\mathcal{L}} t_l$.

Substituting the expression of t_l in t, we obtain our optimal multi-path routing problem, which is aimed at minimizing the shuffle completion time, as follows:

$$\min_{\alpha_{ij}} \quad t \tag{1}$$

$$\text{s.t.} \quad t = \max_{l\in\mathcal{L}} \frac{\sum_i \sum_{j,l\in p(i,j)} d_i\alpha_{ij}}{b_l} \tag{2}$$

$$\sum_{j=1}^{J_i} \alpha_{ij} = 1, \forall i \in \{1,2,\ldots,I\} \tag{3}$$

$$\alpha_{ij} \geq 0, \forall i,j \in \{1,2,\ldots,J_i\}. \tag{4}$$

Constraint (3) indicates that for each aggregated flow, the sum of traffic percentages allocated to each of its paths is 1, representing the nature of ratios. Constraint (4) also stands for the positive nature of ratios.

This problem is equivalent to the following problem:

$$\min_{\alpha_{ij},\lambda} \quad \lambda \tag{5}$$

$$\text{s.t.} \quad \lambda \geq \frac{\sum_i \sum_{j,l\in p(i,j)} d_i\alpha_{ij}}{b_l}, \forall l \in \mathcal{L} \tag{6}$$

$$\text{Constraints (3) and (4).}$$

The intuition is that for any feasible λ, we have $\lambda \geq t = \max_{l\in\mathcal{L}} \frac{\sum_i \sum_{j,l\in p(i,j)} d_i\alpha_{ij}}{b_l}$. It is obvious that the objective achieves the minimum only when $\lambda = t$. Therefore, Problem (1) and Problem (5) are equivalent.

Further, we transform constraint (6) to an equivalent form, which gives the equivalent optimization problem as follows:

$$\min_{\alpha_{ij},\lambda} \quad \lambda \tag{7}$$

$$\text{s.t.} \quad \sum_i \sum_{j,l\in p(i,j)} d_i\alpha_{ij} \leq \lambda b_l, \forall l \in \mathcal{L} \tag{8}$$

$$\text{Constraints (3) and (4).}$$

As observed, the objective is linear, and all the constraints are linear as well. Hence, Problem (7) is a linear programming (LP) problem, which can be solved using efficient LP solvers, such as MOSEK [12].

Selecting the Best One. It is worth noting that there may exist multiple optimal solutions to Problem (1), which result in the same shuffle completion time.

Among these equivalent solutions, we would like to select the one that incurs the minimum overhead with respect to traffic shifting. The principle is that direct paths should be used with priority. Following this principle, we add a penalty function $g(\alpha_{i,j})$ for traffic shifting in the objective

$$g(\alpha_{i,j}) = -\delta \sum_{i,|p(i,j)|=1} \alpha_{ij}, \tag{9}$$

where $|p(i,j)| = 1$ indicates that the jth path of the ith aggregated flow is a direct path, which consists of only one direct link. The expression $\sum_{i,|p(i,j)|=1} \alpha_{ij}$ represents the sum of direct path traffic percentages over all the transfers. δ is a positive parameter which is tuned so that $|g(\alpha_{i,j})|$ is at least two orders of magnitude smaller than λ.

With the penalty function added, we obtain the following optimization problem:

$$\min_{\alpha_{ij},\lambda} \quad \lambda - \delta \sum_{i,|p(i,j)|=1} \alpha_{ij} \tag{10}$$

$$\text{s.t.} \quad \text{Constraints (8), (3) and (4).} \tag{11}$$

The more direct paths used, the smaller the penalty function will be, and thus the smaller the objective. Since the problem is a minimization problem, the solution that incurs the least amount of traffic to be shifted away from their direct paths will be selected, as we desire.

It is obvious that the objective is still linear. Hence, Problem (10) is an LP that can be efficiently solved.

4 Multi-Path Routing and Scheduling for Sharing Jobs

We now consider a general scenario where I aggregated flows from multiple concurrent data analytic jobs are sharing the inter-datacenter network.

If our objective is to minimize the slowest shuffle completion time among all the jobs, then the problem formulation in the previous section is readily applicable to this multi-job scenario. The only difference is that the I aggregated flows are generated from multiple concurrent jobs rather than a single job in the previous section.

However, when we try to further improve the completion time for the next slowest shuffle, the previous formulation is no longer applicable, because of the limit of the default TCP fair sharing it relies upon. In this multi-job sharing scenario, rate assignment plays a significant role in improving the effective utilization. Hence, we modify our previous problem formulation, to involve the flexibility of rate assignment (or *scheduling*), so that an even more flexible solution can be devised to improve shuffle completion times. The additional key notations are presented in Table 2.

TABLE 2
Additional Notations for Multi-Path Routing and Scheduling for Sharing Jobs

Notation	Definition
$\mathcal{P}_i$	Set of all the available paths for ith aggregated flow
$x_i(p)$	Sending rate of the ith aggregated flow along its path $p \in \mathcal{P}_i$
$\lambda_i(p,l)$	Binary indicator on whether link l is along path p of ith aggregated flow, $\lambda_i(p,l) \in \{0,1\}$
$\mathcal{C}_i$	Coflow that aggregated flow i belongs to, which is the set of all its constituting flows.
T	Completion time of the slowest shuffle

4.1 Optimizing the Worst

We first consider the formulation with the objective of accelerating the slowest shuffle, i.e., minimizing the maximal shuffle completion time among all the sharing jobs.

For the aggregated flow $i \in \mathcal{I} = \{1, 2, \ldots, I\}$, the set of all its available paths is denoted as $\mathcal{P}_i$. We allow such a flow to be flexibly split to take any of its paths, at specific sending rates. To be more specific, we use $x_i(p), \forall i \in \mathcal{I}, \forall p \in \mathcal{P}_i$ to represent the sending rate or throughput of the ith aggregated flow along its path p, which is the decision variable in our multi-path routing and scheduling problem. Intuitively, the total throughput achieved along all the paths of flow i is represented as $\sum_{p\in\mathcal{P}_i} x_i(p)$. Hence, its completion time is derived as $d_i / \sum_{p\in\mathcal{P}_i} x_i(p)$, representing the flow size divided by the total throughput.

On each link $l \in \mathcal{L}$, the total rate of all the traversing flows should not exceed its available bandwidth capacity b_l. Let $\lambda_i(p,l) \in \{0,1\}$ represent whether link l is along the path p of the ith aggregated flow. The total throughput of all the traffic along the link l is calculated as $\sum_{i\in\mathcal{I}} \sum_{p\in\mathcal{P}_i} x_i(p)\lambda_i(p,l)$, which is the summation of the throughput over all the possible paths of an aggregated flow that link l belongs to, and the summation over all the aggregated flows. Hence, we obtain the link capacity constraints as follows:

$$\sum_{i\in\mathcal{I}} \sum_{p\in\mathcal{P}_i} x_i(p)\lambda_i(p,l) \le b_l, \quad \forall l \in \mathcal{L}.$$

If we use T to represent the completion time of the slowest shuffle, we have the following optimization problem:

$$\min_{x_i(p)} \quad T \tag{12}$$

$$\text{s.t.} \quad T = \max_{i\in\mathcal{I}} d_i / \sum_{p\in\mathcal{P}_i} x_i(p) \tag{13}$$

$$\sum_{i\in\mathcal{I}} \sum_{p\in\mathcal{P}_i} x_i(p)\lambda_i(p,l) \le b_l, \quad \forall l \in \mathcal{L} \tag{14}$$

$$x_i(p) \ge 0, \quad \forall p \in \mathcal{P}_i, \quad i \in \mathcal{I}. \tag{15}$$

Constraint (13) indicates that T is the maximum completion time among all the shuffles. Constraint (14) is the bandwidth capacity constraint as aforementioned.

With the same justification as in the previous section, we transform Problem (12) to the following equivalent problem:

$$\min_{x_i(p),\gamma} \quad \gamma \tag{16}$$

$$\text{s.t.} \quad \gamma \geq d_i / \sum_{p \in \mathcal{P}_i} x_i(p), \qquad \forall i \in \mathcal{I} \tag{17}$$

Constraints (14) and (15).

If we denote $\mathcal{X} = 1/\gamma$, then by substituting $\gamma = 1/\mathcal{X}$, Problem (16) is rewritten as the following form:

$$\min_{x_i(p),\mathcal{X}} \quad 1/\mathcal{X}$$

$$\text{s.t.} \quad 1/\mathcal{X} \geq d_i / \sum_{p \in \mathcal{P}_i} x_i(p), \qquad \forall i \in \mathcal{I} \tag{18}$$

Constraints (14) and (15).

It is obvious that $\mathcal{X} = 1/\gamma > 0$, thus we can further transform our original problem to the equivalent problem as follows:

$$\max_{x_i(p),\mathcal{X}} \quad \mathcal{X} \tag{19}$$

$$\text{s.t.} \quad 0 < \mathcal{X} \leq \sum_{p \in \mathcal{P}_i} x_i(p)/d_i, \qquad \forall i \in \mathcal{I} \tag{20}$$

Constraints (14) and (15).

It is easy to check that the objective and constraints are both linear. Therefore, our multi-path routing and scheduling problem is equivalent to a linear programming problem. This problem only has $\sum_i |\mathcal{P}_i| + 1$ variables and $\sum_i |\mathcal{P}_i| + I + |\mathcal{L}| + 1$ constraints, where $|\mathcal{P}_i|$ represents the number of paths for aggregated flow i and $|\mathcal{L}|$ denotes the number of inter-datacenter links. Given the limited number of datacenters in practice, both $|\mathcal{P}_i|$ and $|\mathcal{L}|$ are very small. Hence, this LP problem is of small scale and can be efficiently solved.

4.2 Continuously Optimizing the Next Worst

With our primal objective of minimizing the worst completion time efficiently solved as an LP problem (19), we continue to improve the bandwidth utilization to minimize the next worst completion time repeatedly.

The improvement of bandwidth utilization relies on the awareness of the coflow semantics. If two aggregated flows belong to the same job, which means that they come from the same coflow, then the coflow completion time is depending on the slowest one.

Let $\mathcal{C}_i$ represent the coflow that flow i belongs to. $\mathcal{C}_i$ is the set of all its constituting flows. If flow 1 and 2 belong to the same coflow, then we have $\mathcal{C}_1 = \mathcal{C}_2 \supseteq \{1, 2\}$.

After we optimize the worst completion time ($1/\mathcal{X}^*$), we fix the rate assignment for the flow i^* that achieves this completion time. Then, for all the other flows that belong to the same coflow $\mathcal{C}_{i^*}$, they do not need to finish faster than flow i^*, as it does not help improve the coflow completion time. Therefore, we can let them finish at the same time with the slowest flow i^*, by adding the following equality constraint:

$$\sum_{p \in \mathcal{P}_i} x_i(p)/d_i = \mathcal{X}^*, \qquad \forall i \in \mathcal{C}_{i^*}$$

Then in the next round, we try to optimize the completion time of the next worst flow, which belongs to a different coflow and determines the completion time of the coflow

$$\begin{aligned} \max_{x_i(p),\mathcal{X}} \quad & \mathcal{X} \\ \text{s.t.} \quad & 0 < \mathcal{X} \leq \sum_{p \in \mathcal{P}_i} x_i(p)/d_i, && \forall i \in \mathcal{I}' \\ & \sum_{i \in \mathcal{I} - \tilde{\mathcal{I}}} \sum_{p \in \mathcal{P}_i} x_i(p)\lambda_i(p,l) \leq b'_l, && \forall l \in \mathcal{L} \\ & x_i(p) \geq 0, \quad \forall p \in \mathcal{P}_i, \quad i \in \mathcal{I} - \tilde{\mathcal{I}} \\ & \sum_{p \in \mathcal{P}_i} x_i(p)/d_i = \mathcal{X}^*, && \forall i \in \mathcal{C}_{i^*} \end{aligned} \tag{21}$$

where $\mathcal{I}'$ represents the set of flows whose coflow does not have any flow assigned with rate, and $\tilde{\mathcal{I}}$ denotes the set of all the flows whose rates have been assigned in previous rounds. $b'_l = b_l - \sum_{i \in \tilde{\mathcal{I}}} \sum_{p \in \mathcal{P}_i} x_i(p)\lambda_i(p,l)$, which is the updated available bandwidth.

Algorithm 1. Performance-Optimal Multi-Path Rate Assignment for Inter-Datacenter Flows From Sharing Jobs With Max-Min Fairness

Input:
The aggregated flow set $\mathcal{I}$; the coflow set $\mathcal{C}_i, \forall i \in \mathcal{I}$;
The total amount of data to be sent by each aggregated flow i: d_i;
The set of available paths for each aggregated flow i: $\mathcal{P}_i$;
Output:
Rate assignment for each aggregated flow along each path: $x_i(p), \forall i \in \mathcal{I}, \forall p \in \mathcal{P}_i$;
1: Initialize $\mathcal{I}' = \mathcal{I}, \tilde{\mathcal{I}} = \varnothing$;
2: **while** $\mathcal{I}' \neq \varnothing$ **do**
3: Solve the LP Problem (19) to obtain the solution $x^*, \mathcal{X}^*$;
4: Obtain $x_{i^*}(p), \forall p \in \mathcal{P}_{i^*}$, which satisfies $\sum_{p \in \mathcal{P}_{i^*}} x_{i^*}(p)/d_{i^*} = \mathcal{X}^*$;
5: Fix $x_{i^*}(p), \forall p \in \mathcal{P}_{i^*}$; remove them from the variable set (by adding i^* to $\tilde{\mathcal{I}}$);
6: Update the corresponding link bandwidth capacities in Constraints (14);
7: Find other aggregated flows that belong to the same coflow with i^*, add equality constraints: $\sum_{p \in \mathcal{P}_i} x_i(p)/d_i = \mathcal{X}^*, \ \forall i \in \mathcal{C}_{i^*}$;
8: Remove $\mathcal{C}_{i^*}$ from $\mathcal{I}'$;
9: **end while**

In this way, we avoid improving the completion time of the flow that belongs to the same coflow with previously allocated flows. This guarantees that the bandwidth is efficiently used to improve the completion time of a new coflow round by round, rather than improving the flow whose coflow completion time is already determined in previous rounds.

As a result, the optimization problem in the next round is solved over a decreased set of variables with updated constraints and objectives, so that the next worst coflow completion time would be optimized, without impacting the coflow completion time optimized in this round. Such a procedure is repeatedly executed until the completion time of the last coflow has been optimized, as summarized in Algorithm 1. Eventually, all the coflows achieve their best

possible completion times. In this way, max-min fairness among coflows with respect to their completion time performance can be achieved, because no coflow can improve its completion time performance without degrading others. Our algorithm behaves like progressive filling [17]: it allocates bandwidth to coflows in specific proportionality so that their completion times are shortened simultaneously, until one (or more) coflow can no longer increase performance because of traversing the bottleneck link; this coflow gets all its flow rates fixed in this round, and the remaining bandwidth will be allocated to the rest of coflows to get another (or more) coflow fixed similarly in the next round.

Overhead Analysis. The complexity of our algorithm is analyzed as follows. In each iteration, `line 3` solves the LP problem, which only has $\sum_i |\mathcal{P}_i| + 1$ variables and $\sum_i |\mathcal{P}_i| + I + |\mathcal{L}|$ constraints. As mentioned in the previous subsection, both $|\mathcal{P}_i|$ and $|\mathcal{L}|$ are small given the limited number of datacenters in practice. Hence, the problem is of small scale and can be efficiently solved. `Line 4` loops through all the aggregated flows to check whether they have been bottlenecked using the equation. Hence, the complexity of this step is $O(IP)$, where I is the number of aggregated flows and P is the maximum number of paths for each of these flows. `Line 6` updates link bandwidth capacities with a complexity of $O(|\mathcal{L}|)$, intuitively. `Line 7` introduces the complexity of $O(I)$ in identifying the aggregated flows. The maximum possible number of iterations is the total number of coflows, indicating that coflows do not compete with each other at bottleneck links and get their flow rates assigned one by one, which is the rare worst case.

5 Implementation

We have implemented our multi-path routing and scheduling strategy in our software-defined inter-datacenter overlay testbed, which provides a convenient service for Spark jobs for inter-datacenter transfer optimization. In this section, we present the architecture overview of our prototype and elaborate the implementation details of major components.

5.1 Architecture Overview

Fig. 4 gives an architecture overview of the software-defined inter-datacenter overlay network, based on which our transfer optimization service is implemented. The testbed consists of a centralized controller and several proxy nodes distributed in each datacenter, as highlighted with the shaded boxes in the figure.

Adhering to the principle of Software Defined Networking (SDN), the control plane and data plane in our testbed are fully separated. The centralized controller has the global view of the network states, reported from the proxy nodes in each datacenter. Our optimal multi-path routing and scheduling strategy is implemented as a controller module, which informs the decisions to proxy nodes for enforcement.

The data plane consists of the proxy nodes in each datacenter, which are responsible for interacting with Spark jobs and enforcing the decisions instructed by the controller. To be more specific, each proxy is designed and implemented as a high-performance switch at the application layer, which aggregates outgoing flows destined to the same datacenter and forwards them along multiple paths at specific rates according to the controller decisions. The data transport module in the Spark framework is modified to send and receive all inter-datacenter traffic through the proxy nodes via a simple API, to be elaborated later.

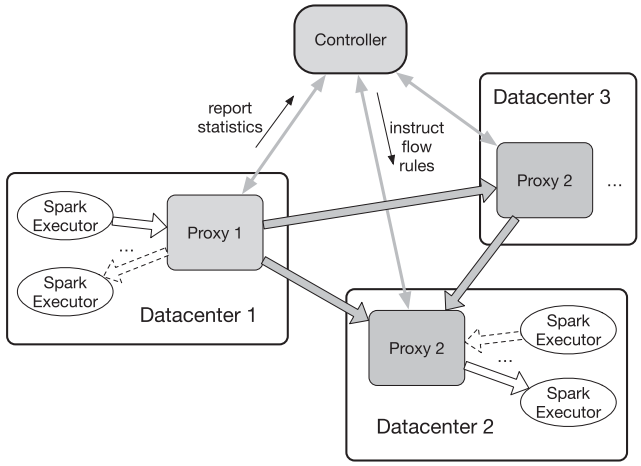


Fig. 4. Overview of our inter-datacenter overlay testbed.

5.2 Implementation of Multi-Path Routing and Scheduling

Our algorithm of multi-path routing and scheduling is implemented in Python, as a pluggable module in the centralized controller. As the input of our algorithm, the inter-datacenter link bandwidth is readily available in the controller, since it can be measured and reported by the proxy nodes periodically. The set of paths for each pair of datacenters is also available, which can be easily obtained by an exhaustive search given a small number of datacenters. To obtain the coflow set and the size of each flow, we modify the `TaskScheduler` module in Spark to report details of network flows in a shuffle. `TaskScheduler` is a component in Spark which decides the placement of all the tasks in a stage. Once the placement decision has been made, we can have the full knowledge (source, destination and the data size) of all the flows in the corresponding shuffle. Our modified `TaskScheduler` can aggregate such coflow information and reports it to the controller. Upon receiving the report, the controller will trigger the execution of our algorithm.

With all the input available, our algorithm will be launched to construct linear programming problems using sparse matrices provided in the `cvxopt` package. The LP problems are then solved with the commercial LP solver in `Mosek` [12]), which is more efficient than the built-in LP solver in `cvxopt`. The calculated routing and scheduling decision for each flow will be conveyed to all the proxy nodes along its assigned path, where the forwarding rules, in the form of `(flowId, nextHops, rates)`, will be installed and enforced in the data plane. Specifically, `nextHops` is an array which specifies the next-hop proxy nodes to relay the traffic, if the flow is to be split along multiple paths. The other array `rates` specifies the assigned bandwidth on the link to the corresponding next hop.

Upon receiving the forwarding rules, the proxy node will enforce the routing and scheduling decisions, by sending flow fragments to the desired next hops in a round-robin manner. Given a next-hop proxy node, it will first re-fragmentize the flow to a fragment, with the size proportional to the assigned bandwidth on the next-hop link. Then, it will relay the flow fragment to the particular next hop, before proceeding to the next `nextHop` in the array. Since fragments from different flows are also served by round-robin on each link, rate assignment is automatically enforced. Note that the flow re-fragmentization is a feature provided by the proxy nodes at the application-layer, which is implemented with little overhead because of the use of zero-copy buffers and smart pointers.

5.3 Integration With Spark

To integrate our transport service with Apache Spark, we need to redirect all its inter-datacenter traffic through our proxy nodes, rather than directly initiating connections between executors. Hence, we have modified the `DataTransferService` module in Spark, which is responsible for data block transfers between executors.

With our modification, if a data block is destined to an executor in another datacenter, it will be forwarded to the local proxy by calling a simple function `publishData()`. For a data block destined to a local executor, the default way will be used to initiate a TCP connection between the executors. In a similar vein, to receive data destined to itself through our transport service, a Spark executor needs to call the function `subscribe()`. Both of these calls are implemented as streaming RPC calls, which are supported by the `gRPC`[1] framework using HTTP/2-based transport at the application layer. Our modification is transparent to Spark users, since no changes need to be made to the data analytic jobs.

6 Performance Evaluation With Real-World Experiments

In this section, we present our evaluation results with a comprehensive set of real-world experiments and extensive simulations, to demonstrate the effectiveness of our multi-path routing and scheduling strategy in optimizing inter-datacenter network transfers and improving job completion times.

Experiment Setup. We deploy our optimal transfer service for a Spark cluster, which spans across 5 datacenters on Google's Cloud Compute Engine, with a total of 17 Virtual Machine (VM) instances. Specifically, 8 VM instances are evenly distributed in Taiwan and Belgium datacenters, 5 VMs are used in N. Carolina, 3 VMs in Oregon and 1 VM in Tokyo. Each VM instance has 2 vCPUs, 13 GB of memory, and a 20 GB SSD of disk storage. One of the VMs in N. Carolina serves as the Spark master, and the rest of the VMs are the Spark workers, running Apache Spark 2.1.0, the latest release as of January, 2017. The controller is deployed on the same VM as the Spark master, in order to minimize their communication overhead. Two proxy nodes are co-located with Spark workers (or executors) in each datacenter.

Methodology. In order to have a fair comparison, we modify the `TaskScheduler` module in Spark, to eliminate the randomness in task placement decisions. In other words, each task in a given workload will be placed on a specific executor across different runs, so that the inter-datacenter traffic generated will be the same. This way, we can fairly compare the performance achieved with different transfer strategies, given the same traffic. We have evaluated the effectiveness of our strategy (Algorithm 1) in optimizing performance for a variety of data analytic workloads, compared with the direct transfer strategy which only uses direct paths. We also compare with two state-of-the-art baselines: *Siphon* [18] and *Rapier* [11], the routing and scheduling strategies designed for inter-datacenter coflows and intra-datacenter coflows, respectively. For a fair comparison, we adapt their algorithms to the inter-datacenter network setting and implement their algorithms in the same software-defined inter-datacenter overlay framework at the application level. Multi-dimensional metrics are measured, including the job completion time, stage completion time and shuffle completion time, to offer a comprehensive comparison.

Machine Learning Workloads. We use three representative machine learning workloads from Spark-Perf Benchmark,[2] which is the official Spark performance test suite created by Databricks.[3] These workloads are:

- *PCA:* Principle Component Analysis.
- *BMM:* Block Matrix Multiplication.
- *Pearson:* Pearson's correlation.

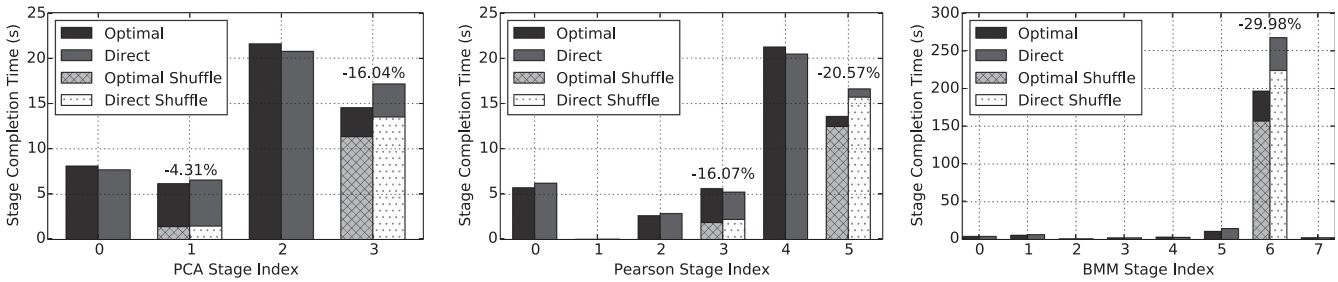
The job completion times (or the application run times, equivalently) achieved by each of the workload with comparing strategies are shown in Fig. 6. As expected, our optimal transfer strategy always outperforms the baseline, achieving a 5.69, 3.18 and 25.24 percent reduction of completion time for *PCA*, *Pearson* and *BMM* workloads, respectively. It is worth noting that *BMM* enjoys the most significant performance improvement from our strategy, with a 25.24 percent reduction in its job completion time. The reason is that it is the most network-intensive workload, with a huge shuffle sending more than 40 GB of data, and the inter-datacenter flows it generates are skew in their sizes.

To allow an in-depth analysis of the reason for such a job-level improvement, we present the completion times for each stage of each workload, achieved with the two strategies, respectively, for comparison. The completion time of each stage is decomposed into the network transfer time (shuffle time) and the computation time, as illustrated in Fig. 5.

As shown in Fig. 5a, a *PCA* job has 4 stages, represented as 0 to 3 along the x-axis. Stage 0 and 2 are computation stages, while stage 1 and 3 consist of shuffles. As the job completion time of *PCA* is dominated by the computation time in stage 0 and 2, the job-level performance does not achieve a significant improvement, as reflected in Fig. 6. However, with respect to the performance of shuffles, our strategy is effective in reducing the shuffle completion

1. https://grpc.io

2. https://github.com/databricks/spark-perf
3. https://databricks.com/



(a) Principal Component Analysis. (b) Pearson's correlation. (c) Block Matrix Multiplication.

Fig. 5. Stage completion times of three machine learning jobs. In each subfigure, the numbers ($-x\%$) are presented above the bars of stages which have network transfers. The negative percentage $-x\%$ indicates that the stage completion time achieved by our Optimal Shuffle is reduced by $x\%$ compared with the Direct Shuffle.

times, by 4.31 percent for stage 1 and 16.04 percent for stage 3, respectively. The similar analysis applies to the *Pearson* job illustrated in Fig. 5b, which is also computation-intensive. The shuffles in both stage 3 and 5 have been accelerated by our strategy, with an improvement of 16.07 and 20.57 percent, respectively.

In contrast, the *BMM* job is network-intensive, so that its job-level performance is significantly influenced by the performance of its shuffles. As demonstrated in Fig. 5c, stage 6, which involves a shuffle, dominates the job completion time. Our optimal strategy achieves a 29.98 percent reduction in the shuffle completion time, which directly contributes to the significant improvement of job-level performance shown in Fig. 6.

To offer a more in-depth examination on the shuffle performance of the *BMM* job, we present the empirical CDF of the shuffle read time for each reduce task, which is the time it takes for a reduce task to finish receiving all its required data, in Fig. 7. Indeed, the set of flows initiated by each reduce task is naturally treated as a coflow, as the shuffle read time is determined by its slowest flow. With multiple reducers in a shuffle, the inter-datacenter links are shared by multiple such coflows. Recalling Algorithm 1, our strategy tries to minimize the worst coflow completion time repeatedly, until all the coflows have achieve their best possible performance. As clearly shown, our optimal strategy successfully reduces the slowest shuffle read time, from 225 s to 160 s. This results in a significant improvement of the shuffle completion time, which is determined by the slowest shuffle read time intuitively. Moreover, the shuffle read times are more evenly distributed, since our strategy always seeks to optimize for the slowest one. In summary, with our optimal routing and scheduling, the inter-datacenter link bandwidth is more efficiently utilized, so that the shuffle phase is accelerated.

Sort. Furthermore, we run the `Sort` application from the HiBench benchmark suite [19], which has only a map stage to sort input data locally and a reduce stage to sort after a heavy shuffle. We generate 2.73 GB of raw input data, which has a skew distribution across the five datacenters. Fig. 8 illustrates the reduce completion times of the sort job achieved by the comparing strategies, respectively. Each of the reduce completion times is further decomposed into the shuffle read time and the task execution time, to offer a more comprehensive comparison. It is easy to observe that our strategy effectively improves the shuffle read time, which contributes to the acceleration of the reduce stage. We further present the empirical CDFs of the shuffle read times achieved by the comparing strategies, respectively, in Fig. 9. Compared with the empirical CDF of the baseline that exhibits a long tail, the shuffle read times in our strategy are more balanced, so that the slowest one is accelerated by 2 s.

Comparison With State-of-the-Art Baselines. As aforementioned, we further present our comparison with *Siphon* [18] and *Rapier* [11] to evaluate the performance of scheduling and routing inter-datacenter coflows. In each run of the experiment, we replay 4 coflows in a testbed deployed across 5 Amazon AWS datacenters distributed across Europe, South-east Asia, and North America. Each coflow has 4 inter-datacenter aggregated flows, the sizes of which range from 200 to 500 MB. The completion times of the

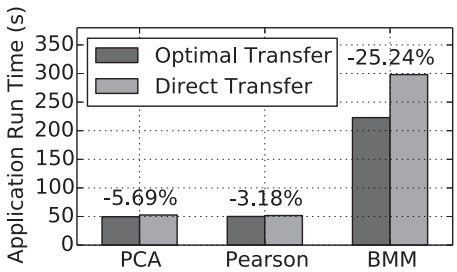


Fig. 6. Job completion times of three machine learning jobs.

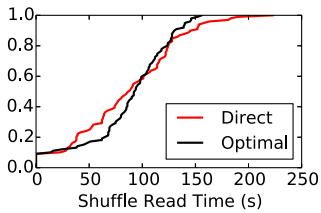


Fig. 7. Empirical CDF of the shuffle read time of the *BMM* job. y-axis is interpreted as the fraction of data.

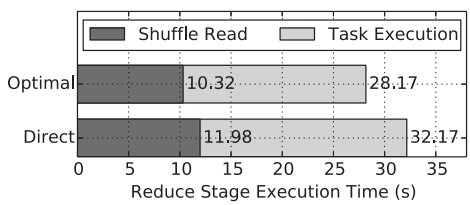


Fig. 8. The execution time of the reduce stage of the *Sort* job, including the shuffle read time and the task execution time.

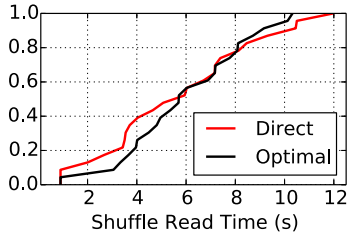


Fig. 9. Empirical CDF of the shuffle read time of the *Sort* job.

slowest coflows achieved in each of the strategies, respectively, are measured. We obtain the speedup over Rapier or Siphon based on the ratio of the worst coflow completion time. We ran the experiment for 21 rounds. The distribution of the speedup statistics is presented in Fig. 11.

Clearly, our strategy outperforms both baselines in accelerating slowest coflows. As compared to Siphon, our scheme shows moderate performance improvement. Because Siphon employs similar scheduling and routing mechanisms for inter-datacenter coflows, the performance is comparable (speedup $< 1.1\times$) in most cases. However, in extreme cases where the inter-datacenter traffic size distribution is highly skewed, the simple heuristic algorithm used in Siphon falls short, and our optimization can achieve a speedup up to $1.351\times$.

When compared with Rapier, our strategy achieves a speedup up to $1.544\times$. In the median case, a 25 percent reduction in the worst coflow completion time can be expected. The reason behind is that Rapier is originally designed for intra-datacenter coflow scheduling, in which the assumption on network topology and bandwidth constraints differs significantly. We further demonstrate the performance improvement over Rapier over 100 runs of simulation. In each run, we simulate 4 coflows in a 4-datacenter scenario. Each coflow has 4 inter-datacenter flows, the sizes of which range from 200 to 500 MB. The completion times of the slowest coflows achieved in each of the strategies, respectively, are measured, based on which we obtain our performance improvement ratio over Rapier as the reduction percentage of the worst coflow completion time. The empirical CDF of the performance improvement ratio is presented in Fig. 10. Clearly, our strategy outperforms Rapier in accelerating the slowest coflow, with an improvement ratio from 5 to 55 percent, depending on the skewness of the inter-datacenter traffic size. Over 50 percent of the 100 runs achieve more than 35 percent of performance improvement.

Scalability. To address the practical concern of our optimization-based strategy, we conduct the following scalability

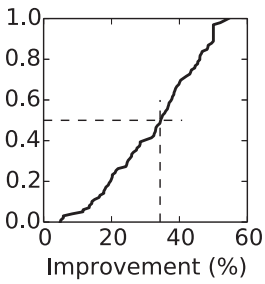


Fig. 10. Empirical CDF of our performance improvement ratio over *Rapier*.

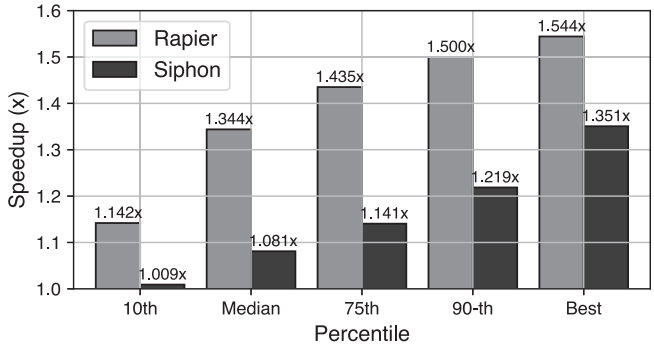


Fig. 11. Speedup achieved by our approach when compared with two baselines, *Rapier* and *Siphon*, respectively.

experiment by running the solver 10 times and collecting the runtime data, given a set of configurations at different scales. More specifically, we consider the setting of x nodes randomly located in y datacenters, where x ranges from 30 to 150 and y is set as 5,6,7, respectively. The number of jobs (or coflows) increases proportionally to the number of cluster nodes. The average runtime (in seconds) and the standard deviation are listed in Table 3. As observed, the runtime of our constructed LP scales reasonably well as the entire Spark cluster grows over 100 nodes. With the number of geographically distributed datacenters increasing, the required solver runtime grows almost quadratically. However, in practice, we envision that seven datacenters are sufficient for most practical and economical deployments.

7 Related Work

Geo-Distributed Data Analytic Jobs. As large volumes of data are increasingly generated globally and stored in geographically distributed datacenters, improving performance of data analytic jobs with geo-distributed input data has received an increasing amount of research attention ([5], [6], [7], [8], [20], [21], [22], [23], [24], [25], *etc.*). G-MR [24] is a Hadoop-based framework that determines schedules for MapReduce job sequences on geo-distributed datasets across multiple datacenters, optimizing for execution time or monetary cost. Hierarchical MapReduce (HMR) [20] and H2F [23] follow a hierarchical scheme, with top layer controller splitting a MapReduce job and partitioning datasets, and bottom-layer managers executing independent MapReduce sub-jobs at different locations. Heintz *et al.* [22] proposed to select mappers and apply shuffle-aware data pushing to meet time constraint, based on prior knowledge of mappers by monitoring the most recent jobs. The assumption of such prior knowledge may not hold, as it is hard to obtain or inaccurate to infer from historical information. Vulimiri *et al.* [5], [6] took the initiative to reduce the total amount of inter-datacenter network traffic when running geo-distributed data analytic jobs. Their solution includes optimizing the query execution plan and the data replication strategy, and also aggressively caching query results. Pixida [7] proposed a graph partition algorithm to divide the directed acyclic graph (DAG) of a job into several parts, each corresponding to a datacenter, so as to minimize the total amount of traffic among these parts.

TABLE 3
The Average Runtime (in Seconds) and the Standard Deviation of Our Solution at Different Scales

# Nodes	5 DCs	6 DCs	7 DCs
30	0.048 ± 0.002	0.119 ± 0.005	0.523 ± 0.032
60	0.156 ± 0.004	0.540 ± 0.005	2.065 ± 0.111
90	0.381 ± 0.029	1.200 ± 0.026	5.223 ± 0.287
120	0.729 ± 0.024	2.166 ± 0.051	9.757 ± 0.627
150	1.130 ± 0.025	3.600 ± 0.086	19.038 ± 1.587

Although the inter-datacenter traffic size is reduced by these efforts, it is not guaranteed that the jobs are accelerated, as job completion times also depend on the available bandwidth of inter-datacenter links. With such an awareness, Iridium [8] proposed an online heuristic to place both input data and tasks across datacenters, under the unrealistic assumption of a congestion-free wide-area network. Flutter [9] removed this assumption and proposed an optimal task placement strategy to optimize the job completion time. Dolev *et al.* [21] presented a more comprehensive survey on the existing works for geo-distributed big-data processing using MapReduce.

However, the efforts above focused on assigning tasks of a single job, which do not account for the general scenario where concurrent jobs have inherent resource competition. Hung *et al.* [26] proposed a greedy scheduling heuristic to determine the execution order of tasks from concurrent jobs in each datacenter, assuming that the placement of tasks are predetermined. Orthogonal to this work, Chen *et al.* [15], [27] designed the optimal placement for tasks of all the sharing jobs with the consideration of fairness.

Different from all these works, we do not reduce the total size of traffic or modify the traffic pattern by specifying the task placement. Instead, given the inter-datacenter traffic, we allow flows to be split along multiple paths and calculate the optimal rate assignment, so that the network transfers from all the sharing jobs achieve their best possible completion times. Our proposed strategy can be integrated as an optimization component in the existing software-defined inter-datecenter transfer framework such as Stemflow [28] and Siphon [18], [29].

Multi-Path Routing and Scheduling for Coflows. Multi-path routing and rate control have attracted much research attention in datacenter networks ([30], [31], *etc.*), among which one category is focused on coflows. RAPIER [11] and Li, *et al.* [10] proposed to jointly consider coflow routing and scheduling, with the objective of minimizing the average coflow completion time. In their solutions, each flow is routed along a single path which is selected from multiple available ones. In contrast, we allow each flow to be split along multiple paths to better utilize the bandwidth. Moreover, our objective is to achieve the best possible performance for all the coflows, with fairness considered. The most important merit of our work is that we have implemented our strategy and evaluated with real coflows, rather than simulated [10] or emulated ones [11].

Performance Optimization for Data Analytic Jobs in a Single Datacenter. There are plenty of existing efforts ([32], [33], [34], *etc.*) related to performance optimization in big data analytic frameworks. They proposed task assignment strategies to improve data locality and fairness [32], [33], or speculation strategies to mitigate the negative impact of stragglers ([34]). With a special focus on the network performance, coflow scheduling strategies ([14], [35], [36], [37], [38], [39], *etc.*) are proposed to minimize the average coflow completion time or to achieve fairness. However, they are all designed for jobs running in a single datacenter, and do not work effectively in the multi-datacenter scenario.

8 Concluding Remarks

In this paper, we have conducted a thorough study on optimizing the inter-datacenter transfers from multiple sharing data analytic jobs whose tasks are geographically distributed across multiple datacenters. Taking the fact into consideration that multiple paths are available for each inter-datacenter flow, we theoretically investigate the multi-path routing problems for inter-datacenter flows from a single job, and those from multiple concurrent jobs, respectively. For the single job scenario, we formulate the multi-path routing problem as a linear programming to be efficiently solved, which minimizes the completion time of the slowest flow. When multiple jobs are considered, our objective is to optimize their shuffle completion times with max-min fairness, which means that the slowest shuffle achieves its fastest completion time and the same for the next slowest one. To achieve this objective, we have designed an algorithm to iteratively optimize the shuffle completion times by solving an updated version of an LP problem. Last but not the least, we have implemented our performance-optimal routing and scheduling strategy, and provided convenient APIs for Spark users to use our service for network performance optimization. Our real-world experiments over Google Cloud with various workloads demonstrated convincing evidence on the effectiveness and advantages of our new strategy.

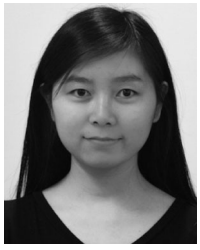
Acknowledgments

This work was supported in part by the Louisiana Board of Regents under Grants LEQSF(2019-22)-RD-A-21 and LEQSF (2021-22)-RD-D-07 and in part by National Science Foundation under Grant OIA-2019511.

References

[1] Microsoft Datacenters, Accessed: May 2021. [Online]. Available: https://www.microsoft.com/en-ca/cloud-platform/global-datacenters
[2] Google Datacenter Locations, Accessed: May 2021. [Online]. Available: https://www.google.com/about/datacenters/inside/locations/
[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[4] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Symp. Netwo. Syst. Des. Implementation*, 2012, Art. no. 2.
[5] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation.*, 2015, pp. 323–336.
[6] A. Vulimiri, C. Curino, P. Godfrey, K. Karanasos, and G. Varghese, "Wanalytics: Analytics for A geo-distributed data-intensive world," in *Proc. Conf. Innovative Data Syst. Res.*, 2015, pp. 1087–1092.

[7] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing data parallel jobs in wide-area data analytics," *VLDB Endowment*, vol. 9, no. 2, pp. 72–83, Oct. 2015.
[8] Q. Pu *et al*., "Low latency geo-distributed data analytics," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, 2015, pp 421–434.
[9] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *Proc. IEEE 35th Annu. Int. Conf. Comput. Commun.*, 2016. pp. 1–9.
[10] Y. Li, S. Jiang, H. Tan, and C. Zhang, "Efficient online coflow routing and scheduling," in *Proc. 17th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, 2016, pp. 161–170.
[11] Y. Zhao *et al*., "RAPIER: Integrating routing and scheduling for coflow-aware data center networks," in *Proc. IEEE Conf. Comput. Commun. INFOCOM*, 2015, pp. 424–432.
[12] E. Andersen and K. Andersen, "The MOSEK interior point optimizer for linear programming: An implementation of the homogeneous algorithm," in *High Perform. Optim.* New York, NY, USA: Springer, 2000, pp. 197–232.
[13] N. McKeown *et al*., "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.
[14] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 443–454.
[15] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," in *Proc. IEEE Conf. Comput. Commun. INFOCOM*, 2017, pp. 1–9.
[16] S. Jain *et al*., "B4: Experience with a globally-deployed software defined WAN," *Proc. ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, 2013.
[17] M. Welzl, *Network Congestion Control: Managing Internet Traffic*. Hoboken, NJ, USA: Wiley, 2005.
[18] S. Liu, L. Chen, and B. Li, "Siphon: Expediting inter-datacenter coflows in wide-area data analytics," in *Proc. USENIX Annu. Techn. Conf.*, 2018, pp. 507–518.
[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the mapReduce-based data analysis," in *Proc. Int. Conf. Data Eng. Workshops*, 2010, pp. 41–51.
[20] Y. Luo and B. Plale, "Hierarchical MapReduce programming model and scheduling algorithms," in *12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2012, pp. 769–774.
[21] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer, "A survey on geographically distributed big-data processing using MapReduce," *IEEE Trans. Big Data*, vol. 5, no. 1, pp. 60–80, Mar. 2019.
[22] B. Heintz, A. Chandra, R. K. Sitaraman, and J. Weissman, "End-to-end optimization for geo-distributed MapReduce," *IEEE Trans. Cloud Comput.*, vol. 4, no. 3, pp. 293–306, Jul.–Sep. 2016.
[23] M. Cavallo, C. Polito, G. D. Modica, and O. Tomarchio, "H2F: A hierarchical hadoop framework for big data processing in geo-distributed environments," in *Proc. 3rd IEEE/ACM Int. Conf. Big Data Comput. Appl. Technol.*, 2016, pp. 27–35.
[24] C. Jayalath, J. Stephen, and P. Eugster, "From the cloud to the atmosphere: Running mapReduce across data centers," *IEEE Trans. Comput.*, vol. 63, no. 1, pp. 74–87, Jan. 2014.
[25] W. Li, X. Yuan, K. Li, H. Qi, X. Zhou, and R. Xu, "Endpoint-flexible coflow scheduling across geo-distributed datacenters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2466–2481, Oct. 2020.
[26] C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *Proc. ACM Symp. Cloud Comput.*, 2015, pp. 111–124.
[27] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," *IEEE Trans. Netw. Sci. Eng.*, vol. 6, no. 3, pp. 488–500, Jul.–Sep. 2019.
[28] S. Liu and B. Li, "Stemflow: Software-defined inter-datacenter overlay as a service," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2563–2573, Nov. 2017.
[29] S. Liu, L. Chen, and B. Li, "Siphon: A high-performance substrate for inter-datacenter transfers in wide-area data analytics," in *Proc. Symp. Cloud Comput.*, 2017, pp. 646–646.
[30] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 266–277.
[31] L. Chen, Y. Feng, B. Li, and B. Li, "Promenade: Proportionally fair multipath rate control in datacenter networks with random network coding," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 11, pp. 2536–2546, Nov. 2019.
[32] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[33] B. Hindman *et al*., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.
[34] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proc. ACM SIGCOMM Conf. Special Int. Group Data Commun.*, 2015, pp. 379–392.
[35] L. Chen, Y. Feng, B. Li, and B. Li, "Towards performance-centric fairness in datacenter networks," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2014, pp. 1599–1607.
[36] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM SIGCOMM Conf. Special Int. Group Data Commun.*, 2015, pp. 393–406.
[37] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proc. ACM Symp. Parallelism Algorithms Architectures*, 2015, pp. 294–303.
[38] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *Proc. IEEE INFOCOM Int. Conf. Comput. Commun.*, 2016, 1–9.
[39] L. Chen, Y. Feng, B. Li, and B. Li, "Efficient performance-centric bandwidth allocation with fairness tradeoff," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1693–1706, Aug. 2018.



Li Chen (Member, IEEE) received the BEng degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2012, the MASc and PhD degrees from the Department of Electrical and Computer Engineering, University of Toronto, in January 2015 and July 2018. She is currently an assistant professor with the Department of Computer Science, School of Computing and Informatics, University of Louisiana at Lafayette. Her research interests include big data analytics, machine learning systems, cloud computing, datacenter networking, resource allocation, and scheduling in networked systems.

Shuhao Liu received the BEng degree from Tsinghua University in 2012. He is currently working toward the PhD degree with the Department of Electrical and Computer Engineering, University of Toronto. His current research interests include datacenter networking and distributed systems.

Baochun Li (Fellow, IEEE) received the PhD degree from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 2000. Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, where he is currently a professor. His research interests include large-scale distributed systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. Since August 2005, he has been the Bell Canada endowed chair in computer engineering. He is a member of the ACM.