Managing Data Transfers in Computer Clusters with Orchestra

Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, Ion Stoica
University of California, Berkeley
{mosharaf, matei, jtma, jordan, istoica}@cs.berkeley.edu

ABSTRACT

Cluster computing applications like MapReduce and Dryad transfer massive amounts of data between their computation stages. These transfers can have a significant impact on job performance, accounting for more than 50% of job completion times. Despite this impact, there has been relatively little work on optimizing the performance of these data transfers, with networking researchers traditionally focusing on per-flow traffic management. We address this limitation by proposing a global management architecture and a set of algorithms that (1) improve the transfer times of common communication patterns, such as broadcast and shuffle, and (2) allow scheduling policies at the transfer level, such as prioritizing a transfer over other transfers. Using a prototype implementation, we show that our solution improves broadcast completion times by up to $4.5\times$ compared to the status quo in Hadoop. We also show that transfer-level scheduling can reduce the completion time of high-priority transfers by $1.7\times$.

Categories and Subject Descriptors

C.2 [**Computer-communication networks**]: Distributed systems—*Cloud computing*

General Terms

Algorithms, design, performance

Keywords

Data-intensive applications, data transfer, datacenter networks

1 Introduction

The last decade has seen a rapid growth of cluster computing frameworks to analyze the increasing amounts of data collected and generated by web services like Google, Facebook and Yahoo!. These frameworks (*e.g.,* MapReduce [15], Dryad [28], CIEL [34], and Spark [44]) typically implement a data flow computation model, where datasets pass through a sequence of processing stages.

Many of the jobs deployed in these frameworks manipulate massive amounts of data and run on clusters consisting of as many as tens of thousands of machines. Due to the very high cost of these clusters, operators aim to maximize the cluster utilization, while accommodating a variety of applications, workloads, and user requirements. To achieve these goals, several solutions have recently been proposed to reduce job completion times [11, 29, 43], accommodate interactive workloads [29, 43], and increase utilization [26, 29]. While in large part successful, these solutions have so far been focusing on scheduling and managing computation and storage resources, while mostly ignoring network resources.

However, managing and optimizing network activity is critical for improving job performance. Indeed, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of the running times of jobs with reduce phases. Existing proposals for full bisection bandwidth networks [21, 23, 24, 35] along with flow-level scheduling [10, 21] can improve network performance, but they do not account for collective behaviors of flows due to the lack of job-level semantics.

In this paper, we argue that to maximize job performance, we need to optimize at the level of transfers, instead of individual flows. We define a *transfer* as the set of all flows transporting data between two stages of a job. In frameworks like MapReduce and Dryad, a stage cannot complete (or sometimes even start) before it receives all the data from the previous stage. Thus, the job running time depends on the time it takes to complete the *entire* transfer, rather than the duration of individual flows comprising it. To this end, we focus on two transfer patterns that occur in virtually all cluster computing frameworks and are responsible for most of the network traffic in these clusters: *shuffle* and *broadcast*. Shuffle captures the many-to-many communication pattern between the map and reduce stages in MapReduce, and between Dryad's stages. Broadcast captures the one-to-many communication pattern employed by iterative optimization algorithms [45] as well as fragment-replicate joins in Hadoop [6].

We propose Orchestra, a global control architecture to manage intra- and inter-transfer activities. In Orchestra, data movement within each transfer is coordinated by a Transfer Controller (TC), which continuously monitors the transfer and updates the set of sources associated with each destination. For broadcast transfers, we propose a TC that implements an optimized BitTorrent-like protocol called Cornet, augmented by an adaptive clustering algorithm to take advantage of the hierarchical network topology in many datacenters. For shuffle transfers, we propose an optimal algorithm called Weighted Shuffle Scheduling (WSS), and we provide key insights into the performance of Hadoop's shuffle implementation.

In addition to coordinating the data movements within each transfer, we also advocate managing *concurrent* transfers belonging to the same or different jobs using an Inter-Transfer Controller (ITC). We show that an ITC implementing a scheduling discipline as simple as FIFO can significantly reduce the average transfer times in

SIGCOMM'11, August 15-19, 2011, Toronto, Ontario, Canada.
Copyright 2011 ACM 978-1-4503-0797-0/11/08 ...$10.00.

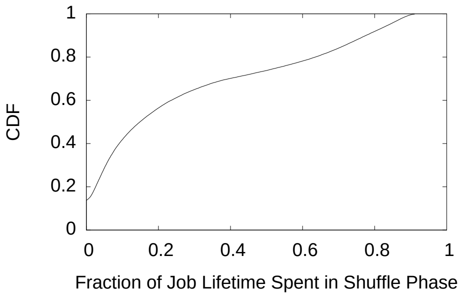


Figure 1: CDF of the fraction of time spent in shuffle transfers in Facebook Hadoop jobs with reduce phases.

a multi-transfer workload, compared to allowing flows from the transfers to arbitrarily share the network. Orchestra can also readily support other scheduling policies, such as fair sharing and priority.

Orchestra can be implemented at the application level and overlaid on top of diverse routing topologies [10,21,24,35], access control schemes [12,22], and virtualization layers [37,38]. We believe that this implementation approach is both appropriate and attractive for several reasons. First, because large-scale analytics applications are usually written using high-level programming frameworks (*e.g.,* MapReduce), it is sufficient to control the implementation of the transfer patterns in these frameworks (*e.g.,* shuffle and broadcast) to manage a large fraction of cluster traffic. We have focused on shuffles and broadcasts due to their popularity, but other transfer patterns can also be incorporated into Orchestra. Second, this approach allows Orchestra to be used in existing clusters without modifying routers and switches, and even in the public cloud.

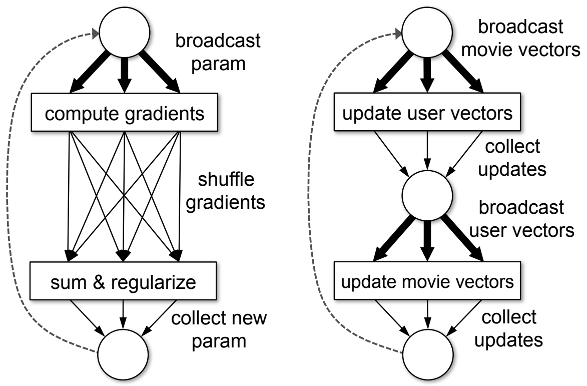
To evaluate Orchestra, we built a prototype implementation in Spark [44], a MapReduce-like framework developed and used at our institution, and conducted experiments on DETERlab and Amazon EC2. Our experiments show that our broadcast scheme is up to $4.5\times$ faster than the default Hadoop implementation, while our shuffle scheme can speed up transfers by 29%. To evaluate the impact of Orchestra on job performance, we run two applications developed by machine learning researchers at our institution—a spam classification algorithm and a collaborative filtering job—and show that our broadcast and shuffle schemes reduce transfer times by up to $3.6\times$ and job completion times by up to $1.9\times$. Finally, we show that inter-transfer scheduling policies can lower average transfer times by 31% and speed up high-priority transfers by $1.7\times$.

The rest of this paper is organized as follows. Section 2 discusses several examples that motivate importance of data transfers in cluster workloads. Section 3 presents the Orchestra architecture. Section 4 discusses Orchestra's inter-transfer scheduling. Section 5 presents our broadcast scheme, Cornet. Section 6 studies how to optimize shuffle transfers. We then evaluate Orchestra in Section 7, survey related work in Section 8, and conclude in Section 9.

2 Motivating Examples

To motivate our focus on transfers, we study their impact in two cluster computing systems: Hadoop (using trace from a 3000-node cluster at Facebook) and Spark (a MapReduce-like framework that supports iterative machine learning and graph algorithms [44]).

Hadoop at Facebook: We analyzed a week-long trace from Facebook's Hadoop cluster, containing 188,000 MapReduce jobs, to find the amount of time spent in shuffle transfers. We defined a "shuffle phase" for each job as starting when either the last map task finishes or the last reduce task starts (whichever comes later) and ending when the last reduce task finishes receiving map outputs. We then measured what fraction of the job's lifetime was spent in this shuffle phase. This is a conservative estimate of the impact of shuffles, because reduce tasks can also start fetching map outputs before all the map tasks have finished.



(a) Logistic Regression (b) Collaborative Filtering

Figure 2: Per-iteration work flow diagrams for our motivating machine learning applications. The circle represents the master node and the boxes represent the set of worker nodes.

We found that 32% of jobs had no reduce phase (*i.e.,* only map tasks). This is common in data loading jobs. For the remaining jobs, we plot a CDF of the fraction of time spent in the shuffle phase (as defined above) in Figure 1. On average, the shuffle phase accounts for 33% of the running time in these jobs. In addition, in 26% of the jobs with reduce tasks, shuffles account for more than 50% of the running time, and in 16% of jobs, they account for more than 70% of the running time. This confirms widely reported results that the network is a bottleneck in MapReduce [10,21,24].

Logistic Regression Application: As an example of an iterative MapReduce application in Spark, we consider Monarch [40], a system for identifying spam links on Twitter. The application processed 55 GB of data collected about 345,000 tweets containing links. For each tweet, the group collected 1000-2000 features relating to the page linked to (*e.g.,* domain name, IP address, and frequencies of words on the page). The dataset contained 20 million distinct features in total. The applications identifies which features correlate with links to spam using logistic regression [25].

We depict the per-iteration work flow of this application in Figure 2(a). Each iteration includes a large broadcast (300 MB) and a shuffle (190 MB per reducer) operation, and it typically takes the application at least 100 iterations to converge. Each transfer acts as a barrier: the job is held up by the slowest node to complete the transfer. In our initial implementation of Spark, which used the the same broadcast and shuffle strategies as Hadoop, we found that communication accounted for 42% of the iteration time, with 30% spent in broadcast and 12% spent in shuffle on a 30-node cluster. With such a large fraction of the running time spent on communication, optimizing the completion times of these transfers is critical.

Collaborative Filtering Application: As a second example of an iterative algorithm, we discuss a collaborative filtering job used by a researcher at our institution on the Netflix Challenge data. The goal is to predict users' ratings for movies they have not seen based on their ratings for other movies. The job uses an algorithm called alternating least squares (ALS) [45]. ALS models each user and each movie as having K features, such that a user's rating for a movie is the dot product of the user's feature vector and the movie's. It seeks to find these vectors through an iterative process.

Figure 2(b) shows the workflow of ALS. The algorithm alter-

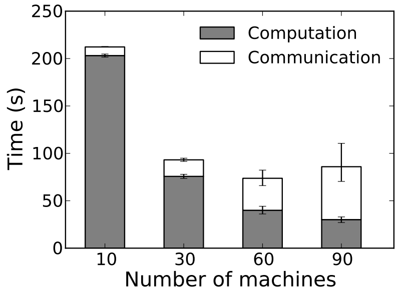


Figure 3: Communication and computation times per iteration when scaling the collaborative filtering job using HDFS-based broadcast.

nately broadcasts the current user or movie feature vectors, to allow the nodes to optimize the other set of vectors in parallel. Each transfer is roughly 385 MB. These transfers limited the scalability of the job in our initial implementation of broadcast, which was through shared files in the Hadoop Distributed File System (HDFS)—the same strategy used in Hadoop. For example, Figure 3 plots the iteration times for the same problem size on various numbers of nodes. Computation time goes down linearly with the number of nodes, but communication time grows linearly. At 60 nodes, the broadcasts cost 45% of the iteration time. Furthermore, the job stopped scaling past 60 nodes, because the extra communication cost from adding nodes outweighed the reduction in computation time (as can be seen at 90 nodes).

3 Orchestra Architecture

To manage and optimize data transfers, we propose an architecture called Orchestra. The key idea in Orchestra is global coordination, both within a transfer and across transfers. This is accomplished through a hierarchical control structure, illustrated in Figure 4.

At the highest level, Orchestra has an Inter-Transfer Controller (ITC) that implements cross-transfer scheduling policies, such as prioritizing transfers from ad-hoc queries over batch jobs. The ITC manages multiple Transfer Controllers (TCs), one for each transfer in the cluster. TCs select a mechanism to use for their transfers (*e.g.,* BitTorrent versus a distribution tree for broadcast) based on the data size, the number of nodes in the transfer, their locations, and other factors. They also actively monitor and control the nodes participating in the transfer. TCs manage the transfer at the granularity of flows, by choosing how many concurrent flows to open from each node, which destinations to open them to, and when to move each chunk of data. Table 1 summarizes coordination activities at different components in the Orchestra hierarchy.

Orchestra is designed for a cooperative environment in which a single administrative entity controls the application software on the cluster and ensures that it uses TCs for transfers. For example, we envision Orchestra being used in a Hadoop data warehouse such as Facebook's by modifying the Hadoop framework to invoke it for its transfers. However, this application stack can still run on top of a network that is shared with other tenants—for example, an organization can use Orchestra to schedule transfers inside a virtual Hadoop cluster on Amazon EC2. Also note that in both cases, because Orchestra is implemented in the framework, users' applications (*i.e.,* MapReduce jobs) need not change.

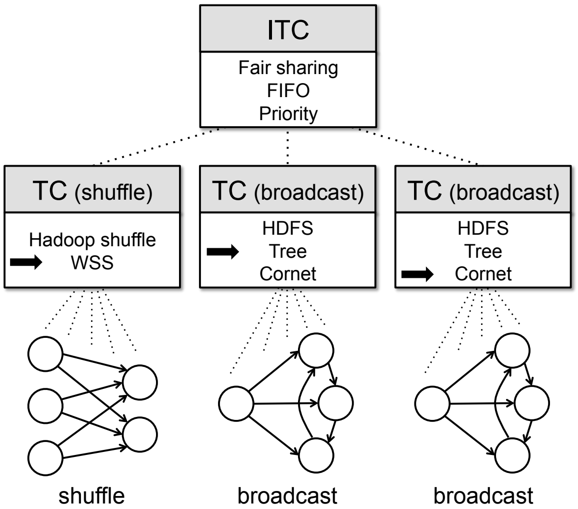


Figure 4: Orchestra architecture. An Inter-Transfer Controller (ITC) manages Transfer Controllers (TCs) for the active transfers. Each TC can choose among multiple transfer mechanisms depending on data size, number of nodes, and other factors. The ITC performs inter-transfer scheduling.

Table 1: Coordination throughout the Orchestra hierarchy.

Component	Coordination Activity
Inter-Transfer Controller (ITC)	- Implement cross-transfer scheduling policies (*e.g.,* priority, FIFO etc.) - Periodically update and notify active transfers of their shares of the network
Transfer Controller (TC)	- Select the best algorithm for a transfer given its share and operating regime
Cornet Broadcast TC	- Use network topology information to minimize cross-rack communication - Control neighbors for each participant
Weighted Shuffle Scheduling TC	- Assign flow rates to optimize shuffle completion time

Since Orchestra can be implemented at the application level, it can be used in existing clusters without changing network hardware or management mechanisms. While controlling transfers at the application level does not offer perfect control of the network or protection against misbehaving hosts, it still gives considerable flexibility to improve job performance. We therefore took this approach as a sweet-spot between utility and deployability.

In the next few sections, we present the components of Orchestra in more detail. First, we discuss inter-transfer scheduling and the interaction between TCs and the ITC in Section 4. Sections 5 and 6 then discuss two efficient transfer mechanisms that take advantage of the global control provided by Orchestra. For broadcast, we present Cornet, a BitTorrent-like scheme that can take into account the cluster topology. For shuffle, we present Weighted Shuffle Scheduling (WSS), an optimal shuffle scheduling algorithm.

4 Inter-Transfer Scheduling

Typically, large computer clusters are multi-user environments where hundreds of jobs run simultaneously [29, 43]. As a result, there are usually multiple concurrent data transfers. In existing clusters, without any transfer-aware supervising mechanism in place, flows from each transfer get some share of the network as allocated by TCP's proportional sharing. However, this approach can lead to extended transfer completion times and inflexibility in enforcing scheduling policies. Consider two simple examples:

- *Scheduling policies:* Suppose that a high-priority job, such as a report for a key customer, is submitted to a MapReduce cluster. A cluster scheduler like Quincy [29] may quickly assign CPUs and memory to the new job, but the job's flows will still experience fair sharing with other jobs' flows at the network level.
- *Completion times:* Suppose that three jobs start shuffling *equal amounts* of data at the same time. With fair sharing among the flows, the transfers will all complete in time $3t$, where t is the time it takes one shuffle to finish uncontested. In contrast, with FIFO scheduling across transfers, it is well-known that the transfers will finish faster on average, at times t, $2t$ and $3t$.

Orchestra can implement scheduling policies at the transfer level through the Inter-Transfer Controller (ITC). The main design question is what mechanism to use for controlling scheduling across the transfers. We have chosen to use weighted fair sharing at the cluster level: *each transfer is assigned a weight, and each congested link in the network is shared proportionally to the weights of the transfers using that link*. As a mechanism, weighted fair sharing is flexible enough to emulate several policies, including priority and FIFO. In addition, it is attractive because it can be implemented at the hosts, without changes to routers and switches.

When an application wishes to perform a transfer in Orchestra, it invokes an API that launches a TC for that transfer. The TC registers with the ITC to obtain its share of the network. The ITC periodically consults a scheduling policy (*e.g.,* FIFO, priority) to assign shares to the active transfers, and sends these to the TCs. Each TC can divide its share among its source-destination pairs as it wishes (*e.g.,* to choose a distribution graph for a broadcast). The ITC also updates the transfers' shares periodically as the set of active transfers changes. Finally, each TC unregisters itself when its transfer ends. Note that we assume a cooperative environment, where all the jobs use TCs and obey the ITC.

We have implemented fair sharing, FIFO, and priority policies to demonstrate the flexibility of Orchestra. In the long term, however, operators would likely integrate Orchestra's transfer scheduling into a job scheduler such as Quincy [29].

In the rest of this section, we discuss our prototype implementation of weighted sharing using TCP flow counts (§4.1) and the scalability and fault tolerance of the ITC (§4.2).

4.1 Weighted Flow Assignment (WFA)

To illustrate the benefits of transfer-level scheduling, we implemented a prototype that approximates weighted fair sharing by allowing different transfers to use different numbers of TCP connections per host and relies on TCP's AIMD fair sharing behavior. We refer to this strategy as Weighted Flow Assignment (WFA). Note that WFA is not a bulletproof solution for inter-transfer scheduling, but rather a proof of concept for the Orchestra architecture. Our main contribution is the architecture itself. In a full implementation, we believe that a cross-flow congestion control scheme such as Seawall [38] would improve flexibility and robustness. Seawall performs weighted max-min fair sharing between applications that use an arbitrary number of TCP and UDP flows using a shim layer on the end hosts and no changes to routers and switches.[1]

In WFA, there is a fixed number, F, of permissible TCP flows per host (*e.g.,* 100) that can be allocated among the transfers. Each transfer i has a weight w_i allocated by the scheduling policy in the ITC. On each host, transfer i is allowed to use $F\lceil\frac{w_i}{\sum w_j}\rceil$ TCP connections, where the sum is over all the transfers using that host. Suppose that two MapReduce jobs (A and B) sharing the nodes of a cluster are both performing shuffles. If we allow each job to use 50 TCP connections per host, then they will get roughly equal shares of the bandwidth of each link due to TCP's AIMD behavior. On the other hand, if we allowed job A to use 75 TCP flows per host and job B to use 25 TCP flows per host, then job A would receive a larger share of the available bandwidth.[2]

Our implementation also divides all data to be transferred into chunks, so that a TC can shut down or launch new flows rapidly when its share in a host changes. In addition, for simplicity, we give transfers a $1.5\times$ higher cap for sending flows than for receiving on each host, so that a TC does not need to micromanage its flows to have sender and receiver counts match up exactly.

We found that our flow count approach works naturally for both broadcast and shuffle. In a BitTorrent-like broadcast scheme, nodes already have multiple peers, so we simply control the number of concurrent senders that each node can receive from. In shuffle transfers, existing systems already open multiple connections per receiver to balance the load across the senders, and we show in Section 6 that having more connections only improves performance.

WFA has some limitations—for example, it does not share a link used by different numbers of sender/receiver pairs from different transfers in the correct ratio. However, it works well enough to illustrate the benefits of inter-transfer scheduling. In particular, WFA will work well on a full bisection bandwidth network, where the outgoing links from the nodes are the only congestion points.

4.2 Fault Tolerance and Scalability

Because the ITC has fairly minimal responsibilities (notifying each TC its share), it can be made both fault tolerant and scalable. The ITC stores only soft state (the list of active transfers), so a hot standby can quickly recover this state if it crashes (by having TCs reconnect). Furthermore, existing transfers can continue while the ITC is down. The number of active transfers is no more than several hundred in our 3000-node Facebook trace, indicating that scalability should not be a problem. In addition, a periodic update interval of one second is sufficient for setting shares, because most transfers last seconds to minutes.

5 Broadcast Transfers

Cluster applications often need to send large pieces of data to multiple machines. For example, in the collaborative filtering algorithm in Section 2, broadcasting an $O(100\,\text{MB})$ parameter vector quickly became a scaling bottleneck. In addition, distributing files to perform a fragment-replicate join[3] in Hadoop [6], rolling out software updates [8], and deploying VM images [7] are some other use cases where the same data must be sent to a large number of machines. In this section, we discuss current mechanisms for implementing broadcast in datacenters and identify several of their limitations (§5.1). We then present Cornet, a BitTorrent-like protocol designed specifically for datacenters that can outperform the default Hadoop implementation by $4.5\times$ (§5.2). Lastly, we present a topology-aware variant of Cornet that leverages global control to further improve performance by up to $2\times$ (§5.3).

5.1 Existing Solutions

One of the most common broadcast solutions in existing cluster computing frameworks involves writing the data to a shared file system (*e.g.,* HDFS [2], NFS) and reading it later from that centralized storage. In Hadoop, both Pig's fragment-replicate join im-

[1] In Orchestra, we can actually implement Seawall's congestion control scheme directly in the application instead of using a shim, because we control the shuffle and broadcast implementations.

[2] Modulo the dependence of TCP fairness on round-trip times.

[3] This is a join between a small table and a large table where the small table is broadcast to all the map tasks.

plementation [6] and the DistributedCache API for deploying code and data files with a job use this solution. This is likely done out of a lack of other readily available options. Unfortunately, as the number of receivers grows, the centralized storage system can quickly become a bottleneck, as we observed in Section 2.

To eliminate the centralized bottleneck, some systems use d-ary distribution trees rooted at the source node. Data is divided into blocks that are passed along the tree. As soon as a node finishes receiving the complete data, it can become the root of a separate tree. d is sometimes set to 1 to form a chain instead of a tree (*e.g.,* in LANTorrent [7] and in the protocol for writing blocks in HDFS [2]). Unfortunately, tree and chain schemes suffer from two limitations. First, in a tree with $d > 1$, the sending capacity of the leaf nodes (which are at least half the nodes) is not utilized. Second, a slow node or link will slow down its entire subtree, which is problematic at large scales due to the prevalence of stragglers [15]. This effect is especially apparent in chains.

Unstructured data distribution mechanisms like BitTorrent [4], traditionally used in the Internet, address these drawbacks by providing scalability, fault-tolerance, and high throughput in heterogeneous and dynamic networks. Recognizing these qualities, Twitter has built Murder [8], a wrapper over the BitTornado [3] implementation of BitTorrent, to deploy software to its servers.

5.2 Cornet

Cornet is a BitTorrent-like protocol optimized for datacenters. In particular, Cornet takes advantage of the high-speed and low-latency connections in datacenter networks, the absence of selfish peers, and the fact that there is no malicious data corruption. By leveraging these properties, Cornet can outperform BitTorrent implementations for the Internet by up to $4.5\times$.

Cornet differs from BitTorrent in three main aspects:

- Unlike BitTorrent, which splits files into blocks and subdivides blocks into small chunks with sizes of up to 256 KB, Cornet only splits data into large blocks (4 MB by default).
- While in BitTorrent some peers (leechers) do not contribute to the transfer and leave as soon as they finish the download, in Cornet, each node contributes its full capacity over the full duration of the transfer. Thus, Cornet does not include a tit-for-tat scheme to incentivize nodes.
- Cornet does not employ expensive SHA1 operations on each data block to ensure data integrity; instead, it performs a single integrity check over the whole data.

Cornet also employs a cap on the number of simultaneous connections to improve performance.[4] When a peer is sending to the maximum number of recipients, it puts further requests into a queue until one of the sending slots becomes available. This ensures faster service times for the small number of connected peers and allows them to finish quickly to join the session as the latest sources for the blocks they just received.

During broadcast, receivers explicitly request for specific blocks from their counterparts. However, during the initial stage, the source of a Cornet broadcast sends out at least one copy of each block in a round-robin fashion before duplicating any block.

The TC for Cornet is similar to a BitTorrent tracker in that it assigns a set of peers to each node. However, unlike BitTorrent, each node requests new peers every second. This allows the TC to adapt to the topology and optimize the transfer, as we discuss next.

5.3 Topology-Aware Cornet

Many datacenters employ hierarchical network topologies with oversubscription ratios as high as 5 [21, 27], where transfer times between two nodes on the same rack are significantly lower than between nodes on different racks. To take network topology into account, we have developed two extensions to Cornet.

CornetTopology In this case, we assume that the network topology is known in advance, which is appropriate, for example, in private datacenters. In CornetTopology, the TC has a configuration database that specifies locality groups, *e.g.,* which rack each node is in. When a receiver requests for a new set of peers from the TC, instead of choosing among all possible recipients (as in vanilla Cornet), the TC gives priority to nodes on the same rack as the receiver. Essentially, each rack forms its individual swarm with minimal cross-rack communication. The results in Section 7.2 show that CornetTopology can reduce broadcast time by 50%.

CornetClustering In cloud environments, users have no control over machine placements, and cloud providers do not disclose any information regarding network topology. Even if the initial placements were given out, VM migrations in the background could invalidate this information. For these cases, we have developed CornetClustering, whose goal is to infer and exploit the underlying network topology. It starts off without any topology information like the vanilla Cornet. Throughout the course of an application's lifetime, as more and more broadcasts happen, the TC records block transfer times between different pairs of receivers and uses a learning algorithm to infer the rack-level topology. Once we infer the topology, we use the same mechanism as in CornetTopology. The TC keeps recalculating the inference periodically to keep an updated view of the network.

The inference procedure consists of four steps. In the first step, we record node-to-node block transfer times. We use this data to construct an $n \times n$ distance matrix $\mathbf{D}$, where n is the number of receiver nodes, and the entries are the median block transfer times between a pair of nodes. In the second step, we infer the missing entries in the distance matrix using a version of the nonnegative matrix factorization procedure of Mao and Saul [33]. After completing the matrix $\mathbf{D}$, we project the nodes onto a two-dimensional space using non-metric multidimensional scaling [31]. Finally, we cluster using a mixture of spherical Gaussians with fixed variance σ^2, and automatically select the number of partitions based on the Bayesian information criterion score [18]. In operational use, one can set σ to the typical intra-rack block transfer time (in our experiments, we use $\sigma = 200$ ms). With enough training data, the procedure usually infers the exact topology and provides a similar speedup to CornetTopology, as we show in Section 7.2.

5.4 Size-Aware Broadcast Algorithm Selection

While Cornet achieves good performance for a variety of workloads and topologies, it does not always provide the best performance. For example, in our experiments we found that for a small number of receivers, a chain distribution topology usually performs better. In such a case, the TC can decide whether to employ one algorithm or another based on the number of receivers. In general, as new broadcast algorithms are developed, the TC can pick the best one to match a particular data size and topology. This ability illustrates the advantage of our architecture, which enables the TC to make decisions based on global information.

6 Shuffle Transfers

During the shuffle phase of a MapReduce job, each reducer is assigned a range of the key space produced by the mappers and must

[4] The default limits for the number of receive and send slots per node are 8 and 12, respectively.

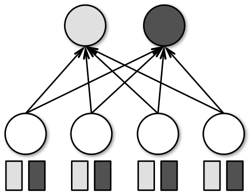
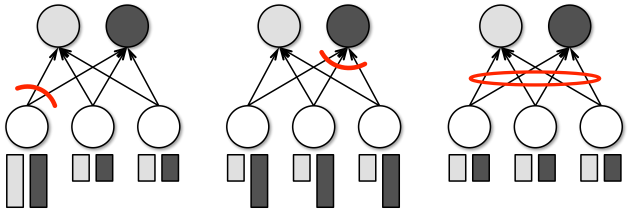


Figure 5: A shuffle transfer. The two receivers (at the top) need to fetch separate pieces of data, depicted as boxes of different colors, from each sender.



(a) Bottleneck at a sender

(b) Bottleneck at a receiver

(c) Bottleneck in the network

Figure 6: Different bottlenecks dictating shuffle performance.

fetch some elements from every mapper. Consequently, shuffle transfers are some of the most common transfer patterns in datacenters. Similar constructs exist in other cluster computing frameworks, like Dryad [28], Pregel [32], and Spark [44]. In general, a shuffle consists of n receivers, $r_1, \ldots, r_n$, and m senders, $s_1, \ldots, s_m$, where each receiver i needs to fetch a distinct dataset d_{ij} from sender j. Figure 5 depicts a typical shuffle.

Because each piece of data goes from only one sender to one receiver, unlike in broadcasts, receivers cannot improve performance by sharing data. The main concern during a shuffle is, therefore, to keep bottleneck links fully utilized (§6.1). We find that the strategy used by systems like Hadoop, where each receiver opens connections to multiple random senders and rely on TCP fair sharing among these flows, is close to optimal when data sizes are balanced (§6.2). There are cases with unbalanced data sizes in which this strategy can perform 1.5× worse than optimal. We propose an optimal algorithm called Weighted Shuffle Scheduling (WSS) to address these scenarios (§6.3). However, situations where WSS helps seem to appear rarely in typical jobs; our main contribution is thus to provide a thorough analysis of this common transfer pattern.

6.1 Bottlenecks and Optimality in Shuffle Transfers

Figure 6 shows three situations where a bottleneck limits shuffle performance and scheduling can have little impact on the overall completion time. In Figure 6(a), one of the senders has more data to send than others (*e.g.,* a map produced more output in a MapReduce job), so this node's link to the network is the bottleneck. Even with the random scheduling scheme in current systems, this link is likely to stay fully utilized throughout the transfer, and because a fixed amount of data *must* flow along this link to finish the shuffle, the completion time of the shuffle will be the same regardless of the scheduling of other flows. Figure 6(b) shows an analogous situation where a receiver is the bottleneck. Finally, in Figure 6(c), there is a bottleneck in the network—for example, the cluster uses a tree topology with less than full bisection bandwidth—and again the order of data fetches will not affect the overall completion time as long as the contended links are kept fully utilized.

These examples suggest a simple optimality criterion for shuffle scheduling: *an optimal shuffle schedule keeps at least one link fully utilized throughout the transfer*. This condition is clearly necessary, because if there was a time period during which a shuffle schedule kept all links less than 100% utilized, the completion time could be lowered by slightly increasing the rate of all flows during that period. The condition is also sufficient as long as unipath routing is used—that is, the data from each sender to each receiver can only flow along one path. In this case, there is a fixed amount of data, D_L, that must flow along each link L, so a lower bound on the transfer time is $\max_L\{D_L/B_L\}$, where B_L is the bandwidth of link L. If any link is fully utilized throughout the transfer, then this lower bound has been reached, and the schedule is optimal. Note that under multipath routing, multiple links may need to be fully utilized for an optimal schedule.

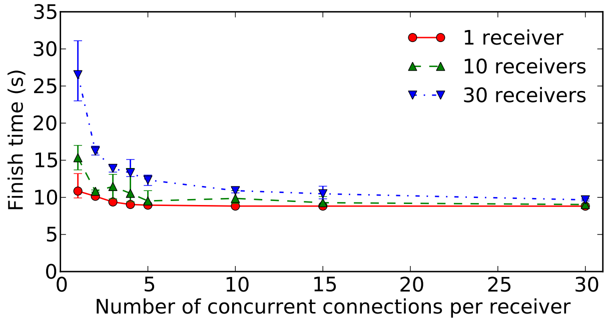


Figure 7: Transfer times for a shuffle with 30 senders and 1 to 30 receivers, as a function of the number of concurrent flows (to random senders) per receiver.

6.2 Load Balancing in Current Implementations

The optimality observation indicates that the links of both senders and receivers should be kept as highly utilized as possible. Indeed, if the amount of data per node is balanced, which is often the case in large MapReduce jobs simply because many tasks have run on every node, then all of the nodes' outgoing links can potentially become bottlenecks. The biggest risk with the randomized data fetching scheme in current systems is that some senders get too few connections to them, underutilizing their links.[5] Our main finding is that having multiple connections per receiver drastically reduces this risk and yields near-optimal shuffle times. In particular, Hadoop's setting of 5 connections per receiver seems to work well, although more connections can improve performance slightly.

We conducted an experiment with 30 senders and 1 to 30 receivers on Amazon EC2, using extra large nodes. Each receiver fetched 1 GB of data in total, balanced across the senders. We varied the number of parallel connections opened by each receiver from 1 to 30. We plot the average transfer times for five runs in Figure 7, with max/min error bars.

We note two trends in the data. First, using a single fetch connection per receiver leads to poor performance, but transfer times improve quickly with even two connections. Second, with enough concurrent connections, transfer times approach 8 seconds asymptotically, which is a lower bound on the time we can expect for nodes with 1 Gbps links. Indeed, with 30 connections per receiver, the overall transfer rate per receiver was 790 Mbps for 30 receivers, 844 Mbps for 10 receivers, and 866 Mbps for 1 receiver, while the best transfer rate we got between any two nodes in our cluster was 929 Mbps. This indicates that randomized selection of senders is within 15% of optimal, and may be even closer because there may

[5] Systems like Hadoop cap the number of receiving connections per reduce task for pragmatic reasons, such as limiting the number of threads in the application. Having fewer connections per receiver can also mitigate incast [41].

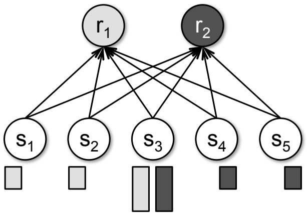


Figure 8: A shuffle configuration where Weighted Shuffle Scheduling outperforms fair sharing among the flows.

be other traffic on EC2 interfering with our job, or a topology with less than full bisection bandwidth.

The improvement in transfer times with more connections happens for two reasons. First, with only one connection per receiver, collisions (when two receivers pick the same sender) can only lower performance, because some senders will be idle. In contrast, with even 2 threads, a collision that slows down some flows may speed up others (because some senders are now sending to only one receiver). Second, with more connections per receiver, the standard deviation of the number of flows to each sender decreases relative to its mean, reducing the effect of imbalances.

6.3 Weighted Shuffle Scheduling (WSS)

We now consider how to optimally schedule a shuffle on a given network topology, where receiver r_i needs to fetch d_{ij} units of data from sender s_j. We aim to minimize the completion time of the shuffle, *i.e.,* the time when the last receiver finishes. For simplicity, we initially assume that the data from a sender/receiver pair can only flow along one path. Under this unipath assumption, the "fully utilized link" condition in Section 6.1 is sufficient for optimality.

We propose a simple algorithm called Weighted Shuffle Scheduling (WSS) that achieves this condition: *allocate rates to each flow using weighted fair sharing, such that the weight of the flow between receiver r_i and sender s_j is proportional to d_{ij}.* To see why this works, consider first a simpler scheduling scheme using progressive filling, where the flow from r_i to s_j is given a rate $t_{ij} = \lambda d_{ij}$ for the largest feasible value of λ. Under this scheme, all the pairwise transfers finish at the same time (because transfer rates are proportional to data sizes for each transfer), and furthermore, at least one link is fully utilized (because we use the largest feasible λ). Therefore, the schedule is optimal. Now, the WSS schedule based on max-min fair sharing with these same weights must finish at least as fast as this progressive filling schedule (because each flow's rate is at least as high as under progressive filling, but may also be higher), so it must also be optimal.

We found that WSS can outperform current shuffle implementations by up to $1.5\times$. In current systems, the flows between senders and receivers experience unweighted fair sharing due to TCP. This can be suboptimal when the flows must transfer different amounts of data. For example, consider the shuffle in Figure 8, where four senders have one unit of data for only one receiver and s_3 has two units for both. Suppose that there is a full bisection bandwidth network where each link carries one data unit per second. Under fair sharing, each receiver starts fetching data at $1/3$ units/second from the three senders it needs data from. After 3 seconds, the receivers exhaust the data on s_1, s_2, s_4 and s_5, and there is one unit of data left for each receiver on s_3. At this point, s_3 becomes a bottleneck, and the receivers take 2 more seconds to transfer the data off. The transfer thus finishes in 5s. In contrast, with WSS, the receivers would fetch data at a rate of $1/4$ units/second from s_1, s_2, s_4 and s_5 and $1/2$ units/second from s_3, finishing in 4s (25% faster).

This discrepancy can be increased in variants of this topology. For example, with 100 senders for only r_1, 100 senders for only r_2, and 100 data units for each receiver on a shared sender, WSS finishes $1.495\times$ faster than fair sharing. Nevertheless, we found that configurations where WSS outperforms fair sharing are rare in practice. If the amounts of data to be transferred between each sender and each reducer are roughly balanced, then WSS reduces to fair sharing. In addition, if there is a single bottleneck sender or bottleneck receiver, then fair sharing will generally keep this node's link fully utilized, resulting in an optimal schedule.

WSS can also be extended to settings where multipath transmissions are allowed. In this case, we must choose transfer rates t_{ij} between receiver i and sender j such that $\min_{i,j}\{t_{ij}/d_{ij}\}$ is maximized. This is equivalent to the Maximum Concurrent Multi-Commodity Flow problem [39].

Implementing WSS Note that WSS requires global knowledge of the amounts of data to transfer in order to set the weight of each flow. WSS can be implemented naturally within Orchestra by having the TC pass these weights to the nodes. In our prototype, the receivers open different numbers of TCP flows to each sender to match the assigned weights. We also tried a variant of where the TC adjusts the weights of each sender/receiver pair periodically based on the amount of data left (in case there is variability in network performance), but found that it provided little gain.

7 Evaluation

We have evaluated Cornet, WSS, and inter-transfer scheduling in the context of Spark and ran experiments in two environments: Amazon's Elastic Compute Cloud (EC2) [1] and DETERlab [5]. On EC2, we used extra-large high-memory instances, which appear to occupy whole physical nodes and had enough memory to perform the experiments without involving disk behavior (except for HDFS-based mechanisms). Although topology information is not provided by EC2, our tests revealed that nodes were able to achieve 929 Mbps in each direction and 790 Mbps during 30 nodes all-to-all communication (Figure 7), suggesting a near-full bisection bandwidth network. The DETERlab cluster spanned 3 racks and was used as ground-truth to verify the correctness of Cornet's clustering algorithm. Our experiments show the following:

- Cornet performs $4.5\times$ better than the default Hadoop implementation of broadcast and BitTornado (§7.1), and with topology awareness in its TC, Cornet can provide further $2\times$ improvement (§7.2).
- WSS can improve shuffle speeds by 29% (§7.3).
- Inter-transfer scheduling can speed up high priority jobs by $1.7\times$, and a simple FIFO policy can improve average transfer response times by 31% for equal sized transfers (§7.4).
- Orchestra reduced communication costs in the logistic regression and collaborative filtering applications in Section 2 by up to $3.6\times$ and sped up jobs by up to $1.9\times$ (§7.5).

Since data transfers act as synchronization steps in most iterative and data-intensive frameworks, capturing the behavior of the slowest receiver is the most important metric for comparing alternatives. We therefore use the *completion time* of the entire transfer as our main performance metric.

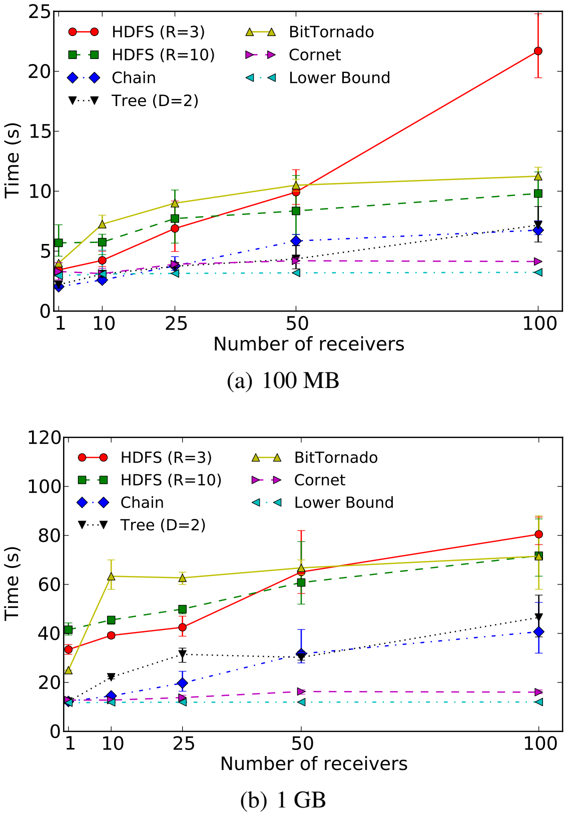
7.1 Comparison of Broadcast Mechanisms

Figure 9 shows the average completion times of different broadcast mechanisms (Table 2) to transfer 100 MB and 1 GB of data to multiple receivers from a single source. Error bars represent the minimum and the maximum observed values across five runs.

We see that the overheads of choking/unchoking, aggressive hashing, and allowing receivers to leave as soon as they are done, fail

Table 2: Broadcast mechanisms compared.

Algorithm	Description
HDFS (R=3)	Sender creates 3 replicas of the data in HDFS and receivers read from them
HDFS (R=10)	Same as before but there are 10 replicas
Chain	A chain of receivers rooted at the sender
Tree (D=2)	Binary tree with sender as the root
BitTornado	BitTorrent implementation for the Internet
Cornet	Approach proposed in Section 5
Theoretical Lower Bound	Minimum broadcast time in the EC2 network (measured to have 1.5 Gbps pairwise bidirectional bandwidth) using pipelined binomial tree distribution mechanism [19]



(a) 100 MB

(b) 1 GB

Figure 9: Completion times of different broadcast mechanisms for varying data sizes.

to take full advantage of the faster network in a datacenter environment and made BitTornado[6] as much as 4.5× slower than the streamlined Cornet implementation.

Cornet scaled well up to 100 receivers for a wide range of data sizes in our experiments. For example, Cornet took as low as 15.4 seconds to complete broadcasting 1 GB data to 100 receivers and remained within 33% of the theoretical lower bound. If there were too few participants or the amount of data was small, Cornet could not fully utilize the available bandwidth. However, as the number of receivers increased, Cornet completion times increased in a much slower manner than its alternatives, which convinces us that Cornet can scale well beyond 100 receivers.

We found structured mechanisms to work well only for smaller scale. Any delay introduced by a straggling internal node of a tree or a chain propagated and got magnified throughout the structure. Indeed, upon inspection, we found that the non-monotonicity of chain and tree completion times were due to this very reason in some experimental runs (*e.g.,* completion time for 25 receivers using a tree structure is larger than that for 50 receivers in 9(b)).

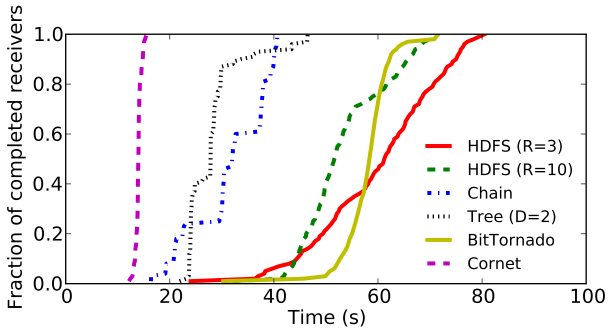


Figure 10: CDF of completion times of individual receivers while transferring 1 GB to 100 receivers using different broadcast mechanisms.

As expected, HDFS-based mechanisms performed well only for small amounts of data. While increasing the number of replicas helps, there is a trade-off between time spent in creating replicas vs. time all the receivers would spend in reading from those replicas. In our experiments, HDFS with 3 replicas performed better than HDFS with 10 replicas when the total number of receivers was less than 50. Overall, HDFS with 3 and 10 replicas were up to 5× and 4.5× slower than Cornet, respectively.

A Closer Look at Per-node Completion Times. We present the CDFs of completion times of individual receivers for each of the compared broadcast mechanisms in Figure 10.

Notice that almost all the receivers in Cornet finished simultaneously. The slight bends at the two endpoints illustrate the receivers ($< 10\%$) that finished earlier or slower than the average receiver. The CDF representing BitTornado reception times is similar to that of Cornet except that the variation in individual completion times is significantly higher and the average receiver is almost 4× slower.

Next, the steps in the CDFs of chain and tree highlight how stragglers slow down all their children in the distribution structure. Each horizontal segment indicates a node that was slow in finishing reception and the subsequent vertical segment indicates the receivers that experienced head-of-line blocking due to a slow ancestor.

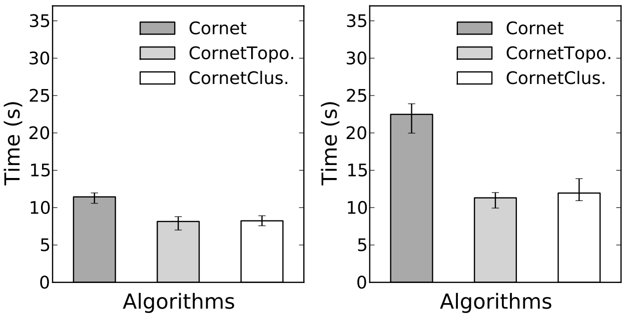
Finally, receivers in HDFS-based transfer mechanism with 10 replicas start finishing slower than those with 3 replicas due to higher replication overhead. However, in the long run, receivers using 10 replicas finish faster because of less reading contention.

The Case for a TC. As evident from Figure 9, the transfer mechanisms have specific operating regimes. In particular, chain and tree based approaches are faster than Cornet for small numbers of nodes and small data sizes, likely because the block sizes and polling intervals in Cornet prevent it from utilizing all the nodes' bandwidth right away. We confirmed this by running another set of experiments with 10 MB (not shown due to lack of space), where tree and chain outperformed the other approaches. In an Orchestra implementation, a TC can pick the best transfer mechanism for a given data size and number of nodes using its global knowledge.

7.2 Topology-aware Cornet

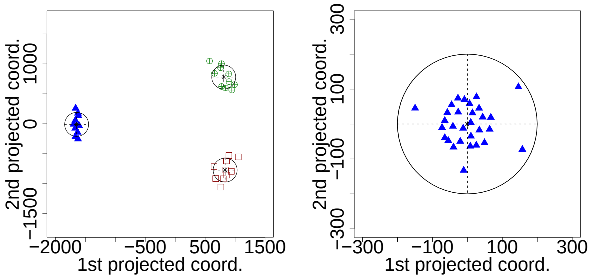
Next, we explore an extension of Cornet that exploits network topology information. We hypothesized that if there is a significant difference between block transfer times within a rack vs. between

[6] We used Murder [8] with a modification that forced every peer to stay in the swarm until all of them had finished.



(a) 100 MB (b) 200 MB

Figure 11: Cornet completion times when the rack topology is unknown, given, and inferred using clustering.



(a) DETERlab (b) Amazon EC2

Figure 12: Two-dimensional, non-metric projection of receiver nodes based on a distance matrix of node-to-node block transfer times. The ellipses represent the inferred clusters. The triangles, squares and circles in (a) represent Rack A, B and C respectively in the DETERlab testbed.

racks, then a topology-aware version of Cornet, which reduces cross-rack communication, will experience improved transfer times. To answer this question, we conducted an experiment on a 31 node DETERlab testbed (1 TC and 30 receivers). The testbed topology was as follows: Rack A was connected to Rack B and Rack B to Rack C. Each rack had 10 receiver nodes. The TC was in Rack B.

We ran the experiment with three TC configurations. The first was the default topology-oblivious Cornet that allowed any receiver to randomly contact any other receiver. The second was CornetTopology, where the TC partitioned the receivers according to Racks A, B, and C, and disallowed communication across partitions. The last one was CornetClustering, where the TC dynamically inferred the partitioning of the nodes based on the node-to-node block transfer times from 10 previous training runs.

The results in Figure 11 show the average completion times to transfer 100 MB and 200 MB of data to all 30 receivers over 10 runs with min-max error bars. Given the topology information (CornetTopology), the broadcasts' completion time decreased by 50% compared to vanilla Cornet for the 200 MB transfer. In 9 out of 10 runs for the 200 MB transfer, the TC inferred the exact topology (see Figure 12(a) for a typical partitioning). Only in one run did the TC infer 5 partitions (splitting two of the racks in half), though this only resulted in a 2.5 second slowdown compared to inferring the exact topology. With the ten runs averaged together, CornetClustering's reduction in completion time was 47%.

We also evaluated Cornet and CornetClustering on a 30 node EC2 cluster. Evaluating CornetTopology was not possible because we could not obtain the ground-truth topology for EC2. The performance of Cornet using inferred topology did not improve over Cornet on EC2 — the algorithm found one cluster, likely due to EC2's high bisection bandwidth (Section 6.2). The projection in Figure 12(b) showed that with the exception of a few outliers (due to congestion), all the nodes appeared to be relatively close to one another and could not be partitioned into well-separated groups.

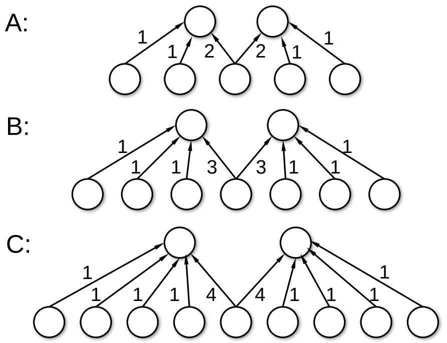


Figure 13: Transfer topologies used in the weighted shuffle experiment. The arrows show the number of units of data sent from each mapper to each reducer.

Table 3: Completion times in seconds for WSS compared to a standard shuffle implementation on the three topologies in Fig. 13. Standard deviations are in parentheses.

Topology	Standard Shuffle	WSS	Speedup	Theoretical Speedup
A	83.3 (1.1)	70.6 (1.8)	18%	25%
B	131 (1.8)	105 (0.5)	24%	33%
C	183 (2.6)	142 (0.7)	29%	38%

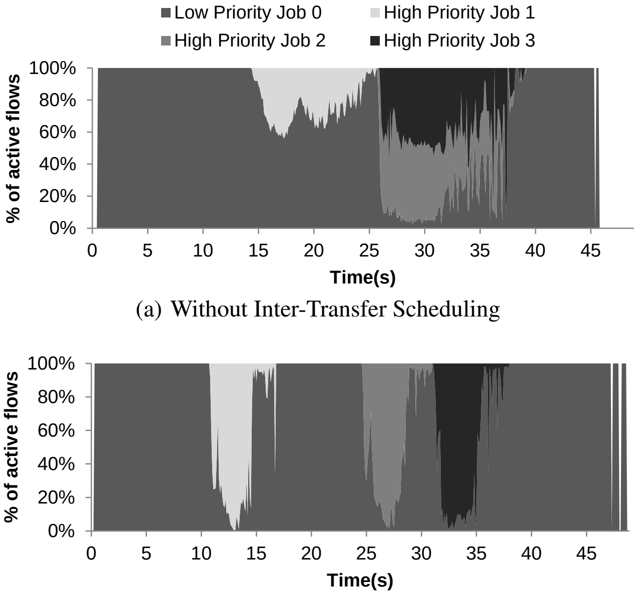
Overall, the results on the DETERlab demonstrate that when there is a sizable gap between intra-rack and inter-rack transfer times, knowing the actual node topology or inferring it can significantly improve broadcast times.

7.3 Weighted Shuffle Scheduling (WSS)

In this experiment, we evaluated the optimal Weighted Shuffle Scheduling (WSS) algorithm discussed in Section 6.3 using three topologies on Amazon EC2. Figure 13 illustrates these topologies, with arrows showing the number of units of data sent between each pair of nodes (one unit corresponded to 2 GB in our tests). Topology A is the example discussed in Section 6.3, where two receivers fetch differing amounts of data from five senders. Topologies B and C are extensions of topology A to seven and nine map tasks.

We ran each scenario under both a standard implementation of shuffle (where each reducer opens one TCP connection to each mapper) and under WSS. We implemented WSS by having each receiver open a different number of TCP connections to each sender, proportional to the amount of data it had to fetch. This allowed us to leverage the natural fair sharing between TCP flows to achieve the desired weights without modifying routers and switches.

We present average results from five runs, as well as standard deviations, in Table 3. In all cases, weighted shuffle scheduling performs better than a standard implementation of shuffle, by 18%, 24% and 29% for topologies A, B and C respectively. In addition, we present the theoretical speedup predicted for each topology, which would be achieved in a full bisection bandwidth network with a perfect implementation of fair sharing between flows. The measured results are similar to those predicted but somewhat



(a) Without Inter-Transfer Scheduling

(b) Priority Scheduling with an Orchestra ITC

Figure 14: Percentage of active flows in the cluster for four different shuffle transfers in two priority classes with and without inter-transfer scheduling.

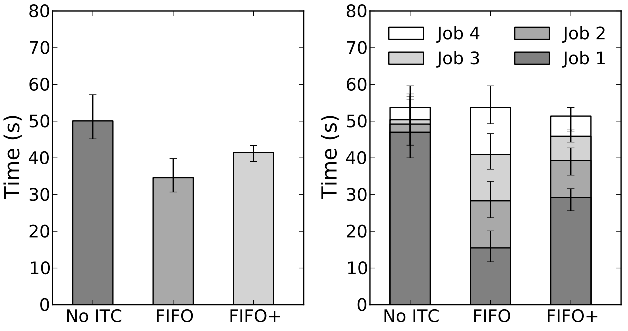
lower because fair sharing between TCP flows is not perfect (*e.g.,* if a node starts 2 GB transfers to several nodes at the same time, these transfers can finish 10-15 seconds apart).

7.4 Scheduling Across Transfers

Cross-Transfer Priority Scheduling In this experiment, we evaluated the average transfer completion times of three different smaller high priority jobs while a low priority larger job was running in the background in a 30-node cluster. The ITC was configured to put an upper limit on the number of flows created by the low priority transfer when at least one high priority job was running. Within the same priority class, the ITC used FIFO to schedule jobs. The larger transfer was shuffling 2 GB data per mapper, while the smaller ones were transferring 256 MB from each mapper. Both the experiments (with or without ITC) in Figure 14 follow similar timelines: the low priority transfer was active during the whole experiment. The first high priority transfer started after at least 10 seconds. After it finished, two more high priority transfers started one after another.

In the absence of any inter-transfer scheduler, the smaller high priority transfers had to compete with the larger low priority one throughout their durations. Eventually, each of the high priority transfers took 14.1 seconds to complete on average, and the larger low priority transfer took 45.6 seconds to finish. When using Orchestra for inter-transfer priority scheduling, the low priority transfer could create only a limited number of flows; consequently, the average completion time for the high priority transfers decreased 43% to 8.1 seconds. The completion time for the background transfer increased slightly to 48.7 seconds.

Cross-Transfer FIFO Scheduling In this experiment, we evaluated the average transfer completion times of four concurrent shuffle transfers that started at the same time in a 30 node cluster. We implemented a strict FIFO policy in the ITC as well as FIFO+, which enforces a FIFO order but gives a small share of the network (3% each) to other transfers as well. The intuition behind FIFO+ is to keep mappers busy as much as possible – if some reducers of the first transfer finish receiving from a particular mapper, it can send



(a) Average completion times (b) Individual completion times

Figure 15: Average completion times of four shuffles with no transfer-level scheduling, FIFO, and FIFO+ (which gives small shares of the network to the transfers later in the FIFO queue).

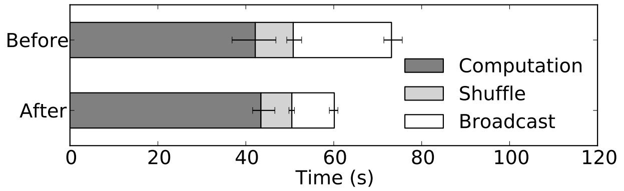


Figure 16: Per-iteration completion times for the logistic regression application before and after using Orchestra.

data to reducers from other transfers. In absence of the ITC, all the flows get their shares of the network as enforced by TCP fair sharing. During each shuffle transfer, each of the 30 reducers received a total of 1 GB data from 30 mappers.

We present the average results from five runs in Figure 15 with max-min error bars. Figure 15(a) shows the average transfer completion times for four transfers. FIFO achieved the best average completion time of 34.6s and was $1.45\times$ faster than TCP fair sharing. The theoretical maximum[7] speedup in this case is $1.6\times$.

The stacked bars in Figure 15(b) show how much longer Job i took to finish its transfer once Job $(i-1)$'s had finished. We observe that the finishing time of the slowest transfer remained the same in both FIFO ordering and TCP fair sharing. However, with FIFO, each transfer actively used the full network for equal amounts of time to complete its transfer and improved the average.

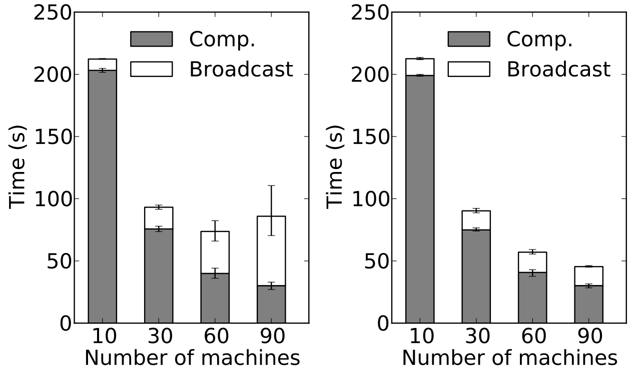
By allowing the latter transfers to have some share of the network, FIFO+ increased the completion time of the first transfer. Meanwhile, it decreased the completion times of all the other transfers in comparison to FIFO. The final outcome is a higher average completion time but lower total completion time than strict FIFO.

7.5 End-to-End Results on Full Applications

We revisit the motivating applications from Section 2 to examine the improvements in end-to-end run times after adopting the new broadcast and shuffling algorithms. In these experiments, we compare the applications before and after adopting Orchestra. In particular, "before" entails running with a HDFS-based broadcast implementation (with the default $3\times$ replication) and a shuffle with 5 threads per receiver (the default in Hadoop). Meanwhile, "after" entails Cornet and shuffle with 30 threads per receiver.

Figure 16 illustrates the breakdown of time spent in different activities in each iteration of Monarch in a 30 node EC2 cluster. We

[7] For n concurrent transfers, theoretical maximum speedup of the average completion time using FIFO over fair sharing is $\frac{2n}{n+1}\times$.



(a) Before

(b) After

Figure 17: Per-iteration completion times when scaling the collaborative filtering application using Orchestra.

see that its communication overhead in each iteration decreased from 42% of the run time to 28%, and iterations finished 22% faster overall. There is a 2.3× speedup in broadcast and a 1.23× speedup in shuffle. The improvements for both broadcast and shuffle are in line with the findings in Sections 7.1 and 6.2.

Figure 17(b) presents the per-iteration completion times for the collaborative filtering job while scaling it up to 90 nodes using Cornet. Unlike the HDFS-based solution (Figure 17(a)), broadcast time increased from 13.4 to only 15.3 seconds using Cornet. As a result, the job could be scaled up to 90 nodes with 1.9× improvement in iteration times. The average time spent in broadcast decreased by 3.6×, from 55.8s to 15.3s, for 90 nodes. These results are in line with Section 7.1 given 385 MB broadcast per iteration.

8 Related Work

Full Bisection Bandwidth Datacenter Networks. A large number of new datacenter network architectures have been proposed in recent years [9, 21, 23, 24, 35] to achieve full bisection bandwidth and improved network performance. However, full bisection bandwidth does not mean infinite bisection bandwidth. Orchestra is still valuable in full bisection bandwidth networks to enable inter-transfer prioritization and scheduling, to balance shuffle transfer rates using WSS, and to speed up broadcasts using Cornet. For example, the experiments in Section 7 show that Orchestra improves job performance even on EC2's network, which appears to have near-full bisection bandwidth.

Centralized Network Controllers. Centralized controllers for routing, access control, and load balancing in the network had been proposed by the 4D architecture [22] and projects like Tesseract [42], Ethane [12], PLayer [30], and Hedera [10]. While PLayer and Ethane focus on access control, our primary objective is application-level performance improvement. The scope of our work is limited to shared clusters and datacenters, whereas 4D, Tesseract, and Ethane are designed for wide-area and enterprise networks. However, unlike Hedera or any of the existing proposals for centralized control planes, we work at the granularity of *transfers* to optimize overall application performance, and not at the packet or flow level.

Performance Isolation in Datacenter Networks. Seawall [38] performs weighted fair sharing among cloud tenants running arbitrary numbers of TCP and UDP flows through a shim layer at the hypervisor using a cross-flow AIMD scheme. It can be leveraged by Orchestra to enforce inter-transfer scheduling policies. However, Seawall itself is not aware of transfer-level semantics.

Scheduling and Management in Data-intensive Applications. A plethora of schemes exist to schedule and manage tasks of data-intensive applications. Examples include fair schedulers for Hadoop [43] and Dryad [29], and Mantri [11] for outlier detection. The core tenet of existing work in this area is achieving data locality to avoid network transfers as much as possible. Mesos [26] provides a thin management layer to allow diverse cluster computing frameworks to efficiently share computation and storage resources, but leaves sharing of network resources to underlying transport mechanisms. Orchestra complements these systems by enabling the implementation of network sharing policies across applications.

One-to-many Data Transfer Mechanisms. Broadcast, multicast, and diverse group communication mechanisms in application and lower layers of the network stack have been studied extensively in the literature. Diot *et al.* provide a comprehensive survey and taxonomy of relevant protocols and mechanisms of distributed multi-point communication in [16]. Cornet is designed for transferring large amounts of data in high-speed datacenter networks.

SplitStream [13] improves network utilization and tackles the bottleneck problem observed in d-ary trees by creating multiple distribution trees with disjoint leave sets. However, it is designed primarily for multimedia streaming over the Internet, where frames can be dropped. Maintaining its structural constraints in presence of failure is complicated as well.

BitTorrent [4] is wildly popular for file-sharing. BitTorrent and similar peer-to-peer mechanisms are in use to distribute planet-scale software updates [20]. However, Murder [8] is the only known BitTorrent deployment inside a datacenter. Antfarm [36] uses a central coordinator across multiple swarms to optimize content distribution over the Internet. Cornet is a BitTorrent-like system that is optimized for datacenters and uses adaptive clustering algorithm in the TC to infer and take advantage of network topologies.

Incast or Many-to-one Transfers. TCP incast collapse is typically observed in barrier-synchronized request workloads where a receiver synchronously receives small amounts of data from a large number of senders [41]. However, incast collapse has been reported in MapReduce-like data-intensive workloads as well [14]. The latter case boils down to a special case of shuffle with only one reducer. With Orchestra, the TC can effectively limit how many senders are simultaneously sending and at what rate to alleviate this problem for data-intensive workloads.

Inferring Topology from Node-to-Node Latencies. Inferring node topology in CornetClustering (Section 5.3) is similar in spirit to inferring network coordinates [17]. These methods could act as a substitute for the non-metric multidimensional scaling step in the CornetClustering procedure.

9 Conclusion

We have argued that multi-node transfer operations have a significant impact on the performance of data-intensive cluster applications and presented an architecture called Orchestra that enables global control both across and within transfers to optimize performance. We focused on two common transfer patterns: broadcasts and shuffles. For broadcasts, we proposed a topology-aware BitTorrent-like scheme called Cornet that outperforms the status quo in Hadoop by 4.5×. For shuffles, we proposed an optimal algorithm called Weighted Shuffle Scheduling (WSS). Overall, our schemes can increase application performance by up to 1.9×. In addition, we demonstrated that inter-transfer scheduling can improve the performance of high-priority transfers by 1.7× and reduce average transfer times by 31%. Orchestra can be implemented at the application level and does not require hardware changes to run in current datacenters and in the cloud.

Acknowledgments

We thank the AMPLab members, the anonymous reviewers, and our shepherd, Yinglian Xie for useful comments on the paper, and Michael Armbrust, Jon Kuroda, Keith Sklower, and the Spark team for infrastructure support. This research was supported in part by gifts from AMPLab founding sponsors Google and SAP, AMPLab sponsors Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp, and VMWare, and by matching funds from the State of California's MICRO program (grants 06-152, 07-010), the National Science Foundation (grants CNS-0509559 and CNS-1038695), the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240, and the Natural Sciences and Engineering Research Council of Canada.

10 References

[1] Amazon EC2. http://aws.amazon.com/ec2.

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

[3] BitTornado. http://www.bittornado.com.

[4] BitTorrent. http://www.bittorrent.com.

[5] DETERlab. http://www.isi.deterlab.net.

[6] Fragment replicate join – Pig wiki. http://wiki.apache.org/pig/PigFRJoin.

[7] LANTorrent. http://www.nimbusproject.org.

[8] Murder. http://github.com/lg/murder.

[9] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. In *SIGCOMM*, pages 51–62, 2010.

[10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[11] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, pages 1–12, 2007.

[13] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[14] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN*, pages 73–82, 2009.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[16] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint communication: A survey of protocols, functions, and mechanisms. *IEEE JSAC*, 15(3):277 –290, 1997.

[17] B. Donnet, B. Gueye, and M. A. Kaafar. A Survey on Network Coordinates Systems, Design, and Security. *IEEE Communication Surveys and Tutorials*, 12(4), Oct. 2010.

[18] C. Fraley and A. Raftery. MCLUST Version 3 for R: Normal mixture modeling and model-based clustering. Technical Report 504, Department of Statistics, University of Washington, Sept. 2006.

[19] P. Ganesan and M. Seshadri. On cooperative content distribution and the price of barter. In *ICDCS*, 2005.

[20] C. Gkantsidis, T. Karagiannis, and M. VojnoviC. Planet scale software updates. In *SIGCOMM*, pages 423–434, 2006.

[21] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[22] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM CCR*, 35:41–54, 2005.

[23] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, pages 63–74, 2009.

[24] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, pages 75–86, 2008.

[25] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2009.

[26] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[27] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[29] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[30] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, 2008.

[31] J. B. Kruskal and M. Wish. Multidimensional Scaling. *Sage University Paper series on Quantitative Applications in the Social Sciences*, 07-001, 1978.

[32] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[33] Y. Mao and L. K. Saul. Modeling Distances in Large-Scale Networks by Matrix Factorization. In *IMC*, 2004.

[34] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A Universal Execution Engine for Distributed Data-Flow Computing. In *NSDI*, 2011.

[35] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, pages 39–50, 2009.

[36] R. Peterson and E. G. Sirer. Antfarm: Efficient content distribution with managed swarms. In *NSDI*, 2009.

[37] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *HotNets 2009*.

[38] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Sharing the data center network. In *NSDI*, 2011.

[39] D. B. Shmoys. *Cut problems and their application to divide-and-conquer*, chapter 5, pages 192–235. PWS Publishing Co., Boston, MA, USA, 1997.

[40] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *IEEE Symposium on Security and Privacy*, 2011.

[41] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM*, pages 303–314, 2009.

[42] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4D network control plane. In *NSDI '07*.

[43] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[45] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *AAIM*, pages 337–348. Springer-Verlag, 2008.