Application-level Flow Scheduling for Efficient Collective Data Transfers
Vijay S. Kumar, Joseph Tucek, Jay J. Wylie, Vijay S. Kumar, Joseph Tucek, Jay J. Wylie

HP Laboratories
HPL-2012-16R1

Abstract:
Collective data transfers among sets of processes over high-bandwidth, low-latency data center networks are an integral part of Big Data computations (e.g., the data shuffle in MapReduce). In this paper, we use a carefully architected microbenchmark that emulates a data shuffle, to gather network traces and perform detailed analysis. The key result of our analysis is that having more than two competing bi-directional flows per node in the transfer reduces throughput by 10%. What this means is that, even at very low cardinality (3- or 4-node shuffle), only 90% of the possible throughput can be achieved when commodity Ethernet-based switches are employed. TCP contention among multiple flows is the reason for the throughput loss experienced by collective data transfers. Though we identify system parameter configurations that minimize such packet losses, we believe application-layer flow management is necessary to circumvent this network-level problem. Towards this end, we designed and implemented a technique, Max2Flows, that generates and orchestrates a schedule of coordinated data exchange stages. Each stage limits the number of competing flows per node to two or fewer, thus avoiding negative network-level effects. Experimental results show that, when incorporated into our microbenchmark, Max2Flows can operate at ≈ 99% of the peak throughput on a 1 Gigabit Ethernet network for small shuffles.
Application-level Flow Scheduling for Efficient Collective Data Transfers

Vijay S. Kumar, Joseph Tucek, Jay J. Wylie
Hewlett-Packard Labs, Palo Alto
Elie Krevat, Gregory R. Ganger
Carnegie Mellon University

Preamble

The following document is an un-edited version of our short paper submission to the 2012 USENIX Annual Technical Conference (ATC). The paper was submitted on January 19, 2012, and was accepted for publication as a short paper on March 22, 2012. However, due to issues in reproducing some of the performance numbers, we withdrew the paper from publication on April 27, 2012. We are making the submitted version of the paper available because the lessons learned tuning a commodity network for shuffle workloads and the application-level flow scheduling technique we developed, Max2Flows, may be of value to others.

The Max2Flows technique was considered interesting and useful by the reviewers. However, the reviewers had concerns about whether such performance drop-offs existed at all for network shuffles. The HP ProCurve switch consistently exhibited the performance drop-offs documented in the following paper for shuffle workloads over a 7-month period from May – December 2011. However, we could not reproduce the performance drop-offs when working on the final copy of our short paper. (We have, however, observed other unexplained throughput losses on one other switch and in a virtualized environment). We continue to run network shuffle experiments on different switches to better understand what could trigger throughput drop-offs. Despite our difficulties reproducing our original results, we believe Max2Flows may be useful in other environments in which throughput losses are observed for all-to-all network patterns.
Application-level Flow Scheduling for Efficient Collective Data Transfers

Vijay S. Kumar, Joseph Tucek, Jay J. Wylie  
*Hewlett-Packard Labs, Palo Alto*

Elie Krevat, Gregory R. Ganger  
*Carnegie Mellon University*

Abstract

Collective data transfers among sets of processes over high-bandwidth, low-latency data center networks are an integral part of Big Data computations (e.g., the data shuffle in MapReduce). In this paper, we use a carefully architected microbenchmark that emulates a data shuffle, to gather network traces and perform detailed analysis. The key result of our analysis is that having more than two competing bi-directional flows per node in the transfer reduces throughput by 10%. What this means is that, even at very low cardinality (3- or 4-node shuffle), only 90% of the possible throughput can be achieved when commodity Ethernet-based switches are employed. TCP contention among multiple flows is the reason for the throughput loss experienced by collective data transfers. Though we identify system parameter configurations that minimize such packet losses, we believe application-layer flow management is necessary to circumvent this network-level problem. Towards this end, we designed and implemented a technique, Max2Flows, that generates and orchestrates a schedule of coordinated data exchange stages. Each stage limits the number of competing flows per node to two or fewer, thus avoiding negative network-level effects. Experimental results show that, when incorporated into our microbenchmark, Max2Flows can operate at \( \approx 99\% \) of the peak throughput on a 1 Gigabit Ethernet network for small shuffles.

1 Introduction

Big Data frameworks continue to grow in popularity and importance as they can easily analyze massive datasets. Examples of Big Data frameworks include MapReduce [6], Hadoop [13], and others [7, 9]. Such frameworks focus on scalability without necessarily paying concern to the efficient use of the underlying data center hardware [8]. Efforts to improve efficiency are primarily predicated on better scheduling of computational jobs and on better management of storage resources [5].

An integral aspect of the dataflow computation model employed in Big Data frameworks is the recurring transfer of large amounts of intermediate data through a sequence of processing stages. For instance, in the map-reduce model, intermediate key-value pairs are shuffled over the network between sets of sender and receiver processes that execute the map and reduce stages of a computation. Chowdhury *et al.* [5] show how jobs on Facebook’s Hadoop cluster spend 33% (worst case: 70%) of their overall execution time on the shuffle phase alone. Shuffle, also known as the all-to-all or many-to-many communication pattern, is just one example of a collective data transfer pattern that places tremendous stress on a cluster’s network resources. As clusters grow in size, the increase in shuffle cardinality (i.e., the number of compute nodes involved in a shuffle data exchange) leads to higher contention for network resources.

The commoditization of network components like switches and the maturation of high-bandwidth Ethernet-based standards have made it affordable for organizations to set up large-scale data centers. In comparison to conventional wide-area networks, data center network architecture is characterized by low latencies (RTTs of the order of hundreds of microseconds), greater reliability and *a priori* knowledge of cluster topologies. Today’s low-cost top-of-the-rack switches have high port densities but proportionately lower buffer queue capacity. Collective data transfers on commodity data center networks are known to experience slowdown effects. Many-to-one transfers have been observed to suffer a 2-3x drop in performance on account of TCP’s incast collapse problem [12]. Our own evaluation of a sort workload over 1 Gigabit Ethernet showed that MapReduce-style shuffles suffer from network skew effects due to TCP’s unfair treatment of the different flows [8]. The TritonSort project [11] reported similar slowdown effects for many-to-many transfers on a 10 Gigabit Ethernet network, and attributed it to thread scheduling issues due to the use...
of small receiver window sizes. All these slowdown effects manifest themselves in the form of lower aggregate throughput for the transfers than the network should be able to deliver. This divergence from idealized network performance for many-to-many transfers motivated us to understand the fundamental reasons behind the throughput loss, when commodity switches are used.

Although shuffling at large scale is hard, interestingly, the most significant drop in performance occurs when scaling from a small-cardinality shuffle (2- to 3-nodes) to a larger-cardinality shuffle (4-nodes, or more). We observe this behavior on switches from different manufacturers. In this paper, we focus on understanding and fixing this low-cardinality shuffle performance problem: (1) We present a detailed evaluation of the data shuffle pattern; this includes empirical results which demonstrate that competing TCP flows cause shuffle inefficiency at low-cardinality, as well as network-level parameter settings that mitigate some performance issues. (2) We describe an application-level flow scheduling scheme called Max2Flows for many-to-many transfers that can circumvent network-level low-cardinality shuffle inefficiency. Max2Flows can achieve \( \approx 99\% \) of the expected throughput for shuffles with cardinality \( \leq 8 \) using commodity switches on a 1 Gigabit Ethernet network.

2 Related Work

Data center networks designed using off-the-shelf Ethernet switches are unable to support the many competing flows of a collective data transfer even when all links operate at full capacity. A huge body of work exists on harnessing the full aggregate network bandwidth of a cluster, including proposals for novel data center network architectures with specialized cluster topologies to provide near-full bisection bandwidth [2]. However, we focus on collective data transfers among sets of nodes that are connected to the same physical switch. To this end, we present only the most relevant efforts, categorized into (i) Alleviating TCP effects, and (ii) Application-layer flow management.

Alleviating TCP effects: Transmission delays occur in data center networks primarily because the operating network characteristics cause TCP to perform poorly. Packet losses occur when many high-bandwidth flows compete – TCP’s congestion control mechanisms and subsequent packet retransmissions on an already loaded network result in a drop in efficiency. Proposed mechanisms to circumvent these negative effects of TCP include: (i) Redesigning the network stack: Variants of TCP [4] provide modified fair sharing and congestion control protocols customized for data center networks, (ii) Use of UDP: Facebook uses memcached, an in-memory data management framework that overcomes TCP’s negative effects by alternatively supporting an enhanced UDP-based network transfer protocol, and (iii) Application-layer reduction of network traffic: Google’s MapReduce framework uses ‘combiner’ functions to move part of the aggregation work to the mappers, thereby reducing the amount of shuffled data.

Application-layer Flow Management: Explicit scheduling of data flows at the application-level has been used to improve transfer performance. In wide-area networks, distributed scheduling is typically carried out at the granularity of network packets or packet bursts. In data center networks, a priori knowledge of cluster topologies can be leveraged to make globally-optimal scheduling decisions [3]. MPI-based clusters use distributed flow scheduling strategies like the pairwise exchange to optimize collective data transfer primitives such as MPI_Broadcast and MPI_Alltoall (an equivalent of the shuffle). In [5], the authors present a case for an application-layer flow management framework for collective data transfers, and show how scheduling at the granularity of entire transfers can be more flexible than at the granularity of individual flows.

We share motivations with the above efforts, but with a specific focus on collective data transfers. We first identify network-level parameter configurations that yield efficient shuffles. Similar to the SNAP profiler [14], we troubleshoot any network performance problems for the shuffle by correlating TCP packet-level statistics with the observed throughput. On our 1 Gbps network, this analysis yielded a configuration that achieves 90% of the peak throughput. This analysis also shows how and why network-level inefficiency occurs when a shuffle has more than two bi-directional flows per node. We leverage this knowledge to develop Max2Flows, an application-level distributed flow management scheme for the data shuffle that achieves throughput close to the wire speed.

Note that we first observed low-cardinality shuffle inefficiency in prior work, but could not explain it at the time [8]. In our prior work, we observed the throughput of shuffles on a compute cluster of 64 nodes (equipped with a 1 Gbps Ethernet network over four Force10 switches) drop significantly when going from two to four nodes. Beyond four nodes, we observed additional gradual slowdown. To be exact, we observed a throughput of 112 MB/s for a 2-node shuffle, 103 MB/s for a 4-node shuffle, and 93 MB/s for a 64-node shuffle. The parameter setting in this paper improves the 2-node shuffle throughput and the Max2Flows technique removes the precipitous throughput drop from 2-node to 4-node shuffles. We do not explore larger shuffles in this paper.
3 Network Shuffle Microbenchmark

Based on our prior work [8], we developed a shuffle microbenchmark that emulates the shuffle phase of a map-reduce style data flow. The microbenchmark provides global throughput statistics and traces the progress of every flow over time. For simplicity, we focus on the case of every node being a sender (mapper) and a receiver (reducer). We made the following design decisions for the microbenchmark:

1. We use a push-based transfer model where senders proactively push data to the receivers. The logically equivalent pull model performs poorly in implementation.

2. Given a shuffle cardinality of $n$ nodes (where each node hosts one mapper and one reducer process), we create a total of $\binom{n}{2} = \frac{n(n-1)}{2}$ connections, where each connection handles bi-directional exchanges between a unique pair of nodes. Using half as many bi-directional links with sockets shared for incoming and outgoing exchanges has better performance than using only uni-directional TCP connections. This decision was inspired by the implementation of the MPI Alltoall primitive in MPICH2 [1].

3. Senders send data from an in-memory buffer, and each receiver silently discards any data it receives. This ensures that the microbenchmark only measures network performance.

The network shuffle microbenchmark is written in C++ and uses POSIX threads to parallelize send and receive tasks across all bi-directional flows. The benchmark measures the average throughput per node (shuffle throughput), starting just after initiating all connections and ending when the last bit of data has been received at all the nodes.

4 Evaluation

To understand the slowdown in shuffle performance and any role that TCP plays in the performance drop, we evaluate our microbenchmark under a host of system parameter configurations. We carry out our experimental evaluation on a commodity cluster consisting of 8 nodes; all nodes reside on a single rack, and are interconnected by a single HP ProCurve 5406zl switch via full-duplex 1 Gbps links. The switch has a 144 MB shared buffer queue, and can be configured with multiple QoS settings. The cluster uses a Linux 2.6.26 kernel with TCP CUBIC for congestion control. The average RTT measured between two nodes on the cluster was $\approx 0.119\text{ms}$. The maximum sustained bandwidth between any pair of nodes (measured using iperf) was 986 Mbps.

4.1 Analysis of Shuffle Slowdown

Using our microbenchmark, we conduct a detailed performance study of the many-to-many data transfer pattern under different network-level parameter settings. Due to space constraints, we limit our discussion in this section to the parameters that had significant impact on shuffle throughput, namely the packet size and segmentation offload.

Packet Size: Ethernet-based clusters today can be configured to use jumbograms or jumbo frames instead of the more common default 1500-byte sized packets. Jumbograms are known to improve throughput on high-bandwidth links by using fewer packets to send the equivalent amount of data. Reducing the proportion of traffic which is headers and reducing CPU load gives modest improvements, but more importantly the increased segment size improves the maximum bandwidth the congestion avoidance can support at a given level of packet loss [10]. We set the MTU for data transfers on our cluster to 9000.

![Figure 1: Jumbograms improve shuffle throughput on a 1 Gbps network (Shuffle size = 4 GB)](image)

Figure 1 shows that configuring our cluster to use jumbograms resulted in an increased shuffle throughput across all cardinalities $\leq 8$. Here, the shuffle size (i.e., the total amount of data being shuffled among the nodes) was fixed at 4 GB as we scaled the shuffle cardinality. The ‘Theoretical peak’ line is the maximum possible bit transmission rate on a full-duplex 1 Gbps network, which cannot be achieved in practice. The maximum throughput achieved using 1500-byte sized packets is 114.1 MB/s (for a 2-node shuffle), and using jumbograms is 119.5 MB/s (for a 3-node shuffle).

The more interesting observation though, is that in both packet size configurations, there is initially a big drop (almost a 10% loss) in throughput for low-cardinality shuffles. When 1500-byte sized packets are used, a loss in throughput of 11 MB/s is observed as we
scale from a 2-node shuffle to a 3-node shuffle. With jumbograms, an identical throughput loss is observed when we scale from cardinality 3 to 4 nodes.

Our analysis of TCP packet-level statistics gathered for each shuffle revealed a direct correlation between the initial big drops in throughput and a spike in the number of TCP segment retransmissions that transpired during the shuffle execution. Table 1 shows an order of magnitude increase in TCP segment retransmits (particularly fast retransmits) at exactly the same shuffle cardinality as the big throughput drop is observed.

<table>
<thead>
<tr>
<th>Packet size</th>
<th>Segment retransmits</th>
<th>Fast retransmits</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTU:1500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shuffle cardinality</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>MTU:9000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1968</td>
<td>2758</td>
</tr>
</tbody>
</table>

This observation merely lends credence to our belief that throughput losses for competing flows in a shuffle are caused by factors that extract undesirable behavior (high number of retransmissions) from TCP congestion control. To rule out packet loss at the end-hosts (i.e., at each sender and receiver node) of the network, we performed experiments with large-sized socket buffer sizes, transmit queue lengths and receiver backlog size limits. Irrespective of any socket-related parameters, for each node, we noted a disparity between the number of TCP segments it sent and received over the course of the shuffle, even while our controlled microbenchmark runs ensure that each node sends and receives the same amount of data.

<table>
<thead>
<tr>
<th>Shuffle cardinality</th>
<th>MTU:1500</th>
<th>MTU:9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP segments received</td>
<td>sent</td>
<td>TCP segments received</td>
</tr>
<tr>
<td>2</td>
<td>1934581</td>
<td>855452</td>
</tr>
<tr>
<td>3</td>
<td>2623607</td>
<td>1244994</td>
</tr>
<tr>
<td>4</td>
<td>2979516</td>
<td>1487993</td>
</tr>
</tbody>
</table>

As seen in Table 2, in both packet size configurations, the average number of segments received by a node over the course of the shuffle is roughly twice the number of segments it sent. Moreover, the number of segments received by a node rightfully drops by ≈6x when jumbograms are used, whereas the number of segments sent by the node drops only by ≈5x. To address this inconsistency in the packetization used on the senders, we configure the segmentation offload parameter.

**Segmentation Offload:** Segmentation offload is a commonly-used technique in networking to increase the outbound throughput of high-bandwidth connections, and is enabled by default in commodity clusters. Here, the segmentation of large buffers on each sender node is offloaded from the CPU to the network interface card in order to reduce CPU and memory overheads. However, the packetization technique used in the CPU could differ from that used in the network interface cards, and there could also be variations across different cards. Hence, the segmentation offload parameter could explain the above disparity in the number of sent and received TCP segments for the shuffle.

Using `ethtool`, we disabled two kinds of offloading — `tso` (TCP segmentation offload) and `gso` (Generic segmentation offload) on the network interface card, thereby forcing segmentation to occur on the CPUs of the senders. We then repeated our earlier set of experiments with this new configuration. Table 3 shows that, with segmentation offload disabled, the average number of TCP segments sent by each node and the number of segments received by each node in the shuffle are within 0.25% of each other in both packet size configurations. We have thus established the grounds for a fairer performance comparison across the two packet sizes.

<table>
<thead>
<tr>
<th>Shuffle cardinality</th>
<th>MTU:1500</th>
<th>MTU:9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP segments received</td>
<td>sent</td>
<td>TCP segments received</td>
</tr>
<tr>
<td>2</td>
<td>1762020</td>
<td>1761673</td>
</tr>
<tr>
<td>3</td>
<td>2478972</td>
<td>2480997</td>
</tr>
<tr>
<td>4</td>
<td>2731970</td>
<td>2732382</td>
</tr>
</tbody>
</table>

Figure 2 shows the effect of disabling the segmentation offload parameter on shuffle throughput. Interestingly,
operate at a throughput of 123.2 MB/s (≈ 99% of the wire speed). We believe this is the maximum throughput that can be practically achieved on a 1 Gbps network (the remaining can be attributed to the various TCP, IP, and Ethernet level headers and checksums); this equals the maximum sustained bandwidth per node pair as reported by iperf. However, the big drop in throughput thereafter ensures that larger shuffles (cardinality ≥ 4) do not achieve the optimal network performance.

Table 4: Network traffic overheads for the shuffle (with segmentation offload disabled)

<table>
<thead>
<tr>
<th>Shuffle cardinality</th>
<th>% of extra segments</th>
<th>#acks with no data payload</th>
<th>#Segment retransmits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.76</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.69</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>185.15</td>
<td>2428</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>3027.4</td>
<td>4372</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>3635.3</td>
<td>5531</td>
</tr>
<tr>
<td>7</td>
<td>25</td>
<td>39929.9</td>
<td>6617</td>
</tr>
<tr>
<td>8</td>
<td>27</td>
<td>44684.4</td>
<td>7436.6</td>
</tr>
</tbody>
</table>

Table 4 provides TCP packet-level statistics which suggest that smaller shuffles (i.e., 2- and 3-nodes) closely achieve the peak throughput since they suffer only minimal packet losses, whereas larger cardinality shuffles (4-nodes, or more) experience considerable packet loss. Here, the ‘% of extra segments’ is calculated as the average number of additional segments transferred per node normalized over the theoretical minimum number of segments transferred for the given packet size. The table also lists the number of acknowledgments that could not be piggybacked with useful data payload, and the number of segment retransmits. In particular, note that the 2- and 3-node shuffles entail no segment retransmits and minimal network overheads.

Packet losses in commodity switches result in inefficient data shuffles at low cardinalities. With the appropriate configuration of network-level parameters such as packet size and segmentation offload, shuffles operate at roughly 90% of the network throughput on our cluster. Our key observation though, one we exploit below, is that many distinct 2- and 3-node shuffles can operate at peak throughput simultaneously.

4.2 Max2Flows: Shuffle Scheduling

Further investigation of packet loss for data shuffles led us to our key insight: more than two competing bi-directional flows per node saturates the switch port that the node connects to, thereby leading to packet losses. To reach this conclusion, we had to rule out that the switch had an upper limit on the total number of flows it could effectively support. To determine that the switch could support the desired number of flows, we compared a 4-node shuffle (6 bi-directional flows among four nodes) with two 3-node shuffles (a total of 6 bi-directional flows, 3 bi-directional flows for each clique of three nodes). The 4-node shuffle suffers a throughput drop, whereas the two simultaneous 3-node shuffles do not. The switch is able to handle large numbers of flows and large data exchanges, so long as they arise only from multiple lower-cardinality (2 and 3-node) shuffles.

Figure 3 is an illustration of the data flows for shuffles of cardinality 2, 3 and 4. While 2- and 3-node shuffles operate at 986 Mbps on our cluster, the 4-node shuffle suffers from packet loss and gets only 890 Mbps. The reason why packet loss occurs only for the higher cardinality shuffle can be explained using the shuffle communication pattern—the 2- and 3-node shuffles have a maximum of two competing bi-directional flows per node, whereas the 4-node shuffle has more than two. The switch ports are unable to support more than two competing high-bandwidth flows directed at the same receiver node. Thus, larger shuffles with more than two competing bi-directional flows per node suffer packet loss that leads to the observed 10% drop in throughput.

We believe that this kind of switch port saturation is a common behavior across many off-the-shelf Ethernet switches in the presence of competing high-bandwidth flows even for low cardinality shuffles. To circumvent this limitation, we propose an application-level distributed flow management scheme for higher-cardinality data shuffles called Max2Flows. Max2Flows orchestrates the many flows of a shuffle in such a way that the switch is handling a maximum of two bi-directional flows per node at all times during the shuffle execution.

Given a shuffle of cardinality $n > 3$, Max2Flows generates a sequential schedule of coordinated data exchange stages, such that (i) the union of data exchanges from all stages constitute the entire set of flows for the $n$-node shuffle, and (ii) the maximum number of bi-directional flows per node within each stage is two. As a result, data exchanges within each stage can operate at close to the wire speed, as no packet loss occurs at the switch ports. Node pairs constituting flows
that are scheduled to exchange data within a stage, exchange all their data within that stage. The minimum number of stages required for an $n$-node shuffle schedule is bounded by $\left\lfloor \frac{n}{2} \right\rfloor$. Figure 4 illustrates the operation of the Max2Flows scheme. Specifically, it shows how Max2Flows decomposes a 9-node shuffle into 4 stages of data exchanges, each of which restricts the number of flows per node to two or fewer and so avoids excessive packet loss.

Figure 4: Max2Flows 9-node shuffle schedule

We incorporated Max2Flows into our microbenchmark and repeated our earlier experiments using our best parameter configuration from Section 4. Figure 5 shows that with the application-level flow scheduling in place, higher-cardinality shuffles (4-nodes to 7-nodes) operate at roughly 980-985 Mbps on a 1 Gbps Ethernet network. Max2Flows circumvents the 10% throughput drop experienced when scaling up from 2- or 3-node shuffles to shuffles of 4-nodes, or more.

5 Discussion

We want to further evaluate the Max2Flows algorithm. In particular, we want to evaluate it for larger shuffle cardinalities, richer network topologies, and more diverse hardware (beyond the HP ProCurve and Force10 switches). We also want to better understand how sensitive Max2Flows is to network or data skew since each stage of the scheduled exchanges blocks on the prior stage’s completion. Finally, we want to incorporate the Max2Flows technique into a Big Data framework such as Hadoop (though it’s pull based data exchange may not be amenable).

References

[12] Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In SIGCOMM 2009, ACM.