BAShuffler: Maximizing Network Bandwidth Utilization in the Shuffle of YARN

Feng Liang
Department of Computer Science
The University of Hong Kong
loengf@connect.hku.hk

Francis C.M. Lau
Department of Computer Science
The University of Hong Kong
fcmlau@cs.hku.hk

ABSTRACT

YARN is a popular cluster resource management platform. It does not, however, manage the network bandwidth resources which can significantly affect the execution performance of those tasks having large volumes of data to transfer within the cluster. The shuffle phase of MapReduce jobs features many such tasks. The impact of underutilization of the network bandwidth in shuffle tasks is more pronounced if the network bandwidth capacities of the nodes in the cluster are varied.

We present BAShuffler, a bandwidth-aware shuffle scheduler, that can maximize the overall network bandwidth utilization by scheduling the source nodes of the fetch flows at the application level. BAShuffler can fully utilize the network bandwidth capacity in a max-min fair network. The experimental results for a variety of realistic benchmarks show that BAShuffler can substantially improve the cluster's shuffle throughput and reduce the execution time of shuffle tasks as compared to the original YARN, especially in heterogeneous network bandwidth environments.

Keywords

YARN; MapReduce; Shuffle; Network Scheduling

1. INTRODUCTION

YARN (Hadoop Version 2) [13] is a fault-tolerant, highly reliable and scalable distributed computing platform for big data processing. YARN exercises a fine-grain control over a cluster's resources, including memories and CPU cores but not the network bandwidths that are available to the cluster nodes. In reality, many tasks running on YARN may need to transfer non-trivial amounts of data among themselves, which happens when executing the shuffle phase of MapReduce [7] and Spark [14]. The performance of these tasks can be largely affected by the network bandwidths allocated to them (by some scheduling of the communication flows). It has been reported that for "shuffle-heavy" jobs, the data shuffling time can take up to as much as 70% of the overall execution time [5]. Optimizing the shuffle performance is thus of paramount importance for these shuffle-heavy jobs.

In MapReduce, the worker of a reduce task needs to fetch the map outputs from a set of mappers via a limited number of TCP flows. The shuffle phase in YARN by default would try to evenly distribute the load on the network by randomly selecting the source node (which corresponds to one or more pending flows) when such a fetch occurs. If the links connecting all the nodes in the cluster are more or less equal in terms of bandwidth and if the number of network connections is large, this random source selection (RSS) policy could prevent some nodes/links from becoming a bottleneck. Obviously, however, without monitoring the ongoing connection allocations in the cluster, and based on which to select a source node to schedule its flows, the RSS approach cannot offer any bandwidth guarantee to the selected flows. In the case where the network is heterogeneous in terms of its links' capacities, RSS would very likely lead to suboptimal performance in scheduling all the flows in a shuffle.

We propose BAShuffler, a network bandwidth aware shuffle scheduler, that can maximize the network bandwidth utilization during the shuffle phase. BAShuffler operates at the application level, without changing the underlying network and the MapReduce interfaces. BAShuffler applies the Partially Greedy Source Selection (PGSS) method to select the appropriate source nodes that can maximize the network bandwidth utilization. PGSS estimates the bandwidth utilization via the notion of max-min fairness in TCP communication. We use examples to illustrate how PGSS works and can increase the bandwidth utilization in different scenarios, for both homogeneous and heterogeneous networks. Our experiment results on a physical cluster show that BAShuffler can significantly increase the shuffle throughput and reduce the total job completion time by up to 29% for shuffle-heavy jobs as compared to the original YARN.

2. APPLICATION-LEVEL SCHEDULING

2.1 Application- vs. Network-Level Design

To improve shuffle performance, a solution can be devised to operate at the network level or the application level. At the network level, ideas such as performance isolation [8] and fair sharing of network resources [12] can provide performance guarantees for the shuffle fetch flows. However, as the flows belonging to one shuffle are correlated in semantics and the shuffle phase cannot finish until its last flow finishes, optimization by scheduling the network based on the granule

HPDC'16, May 31-June 04, 2016, Kyoto, Japan
© 2016 ACM. ISBN 978-1-4503-4314-5/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2907294.2907296

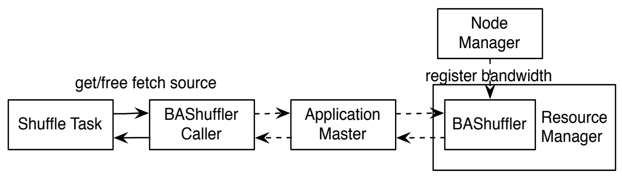


Figure 1: Architecture of BAShuffler

of individual flows may not always lead to improved shuffle performance.

The "coflow" model [6] was proposed to allow scheduling the network based on the granule of a collection of application-level correlated flows. But nevertheless, neither coflow nor any other pure network-level model can actually possess any knowledge about the runtime status of the shuffle phase due to information the gap between the network level and the application level. The application level can only create a limited number of fetch flows at a time (5 per reduce task by default) due to system limits, and the remaining map outputs are left pending until there are available fetch workers later, which is unknown to the network level (or any coflow there). Minimizing the completion time of a coflow is NOT the same thing as minimizing the shuffle completion time.

To obtain the optimal scheduling solution that minimizes the shuffle completion time, the scheduler needs to consider both the application-level runtime status (all the available map outputs and their target destinations) and network-level information (the network fabric, routing, bandwidth allocation, etc.). However, it is too costly to implement such a scheduler because it will need to gather/distribute a large amount of information from/to both the application level and the network level. The overhead of the cross-layer communication between the application level and the network level could be prohibitive.

Application-level shuffle scheduling has the advantage that it can readily obtain the true runtime status of the shuffle. Although it cannot do anything to improve the operation of the underlying network, it can observe and predict the behavior and performance of the network, and then schedule the shuffle flows accordingly based on these observations and the predicted values (e.g., by using max-min fairness in the TCP network) to obtain a near-optimal solution.

2.2 Max-Min Fairness in TCP

The max-min fair (MMF) allocation behavior of TCP communication is the converged state achieved by the AIMD (Additive Increase, Multiplicative Decrease) congestion control algorithm used by TCP [9]. The MMF policy of TCP has been extensively analyzed and verified in the literature [4]. Although it cannot accurately model the exact behavior of TCP communication, the MMF model is acceptable and appropriate for approximating the network behavior, and can lead to useful conclusions in various application settings. By the notion of MMF, the current bandwidth allocated to each TCP flow can be estimated, given the knowledge of the topology, the capacities of the links and the routing paths of the flows.

With the recent progress in research on full bisection bandwidth topologies [2, 11], it is reasonable to simplify the datacenter fabric as a non-blocking switch [5, 3]. In this case, the bottleneck links of the flows would then lie in the access layer which is directly connected to the nodes. When working out the MMF allocation of the TCP flows, the network topology and the paths of the data flows can be ignored, and only the bandwidth capacities of the links in the access layer and the source and destination nodes of the TCP flows need to be considered. The non-blocking switch abstraction largely simplifies the estimation of the MMF bandwidth allocation. In the rest of the paper, we hold the assumption that all bottleneck links are in the access layer.

Algorithm 1: Partially Greedy Source Selection

```
Input  : Sources; Pattern: the sources and destinations
         the allocated flows in the cluster
Output: Selected
Heaviest ← the heaviest-loaded source nodes in Sources
MaxBandwidth ← 0
foreach $Source \in Heaviest$ do
    Util ← the MMF bandwidth utilization of the whole
    cluster after adding Source to Pattern
    if $Util > MaxBandwidth$ then
        MaxBandwidth ← Util
        Selected ← Source
    end
end
add Selected to Pattern
update the load count of Selected
```

3. BASHUFFLER

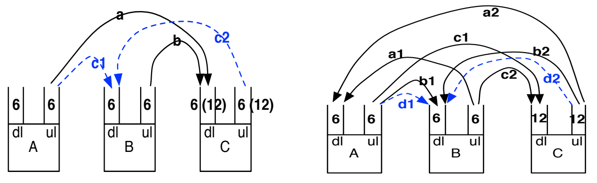
3.1 Architecture of BAShuffler

BAShuffler is implemented and embedded in YARN, and its architecture is shown in Fig. 1. When a shuffle task needs to schedule a new fetch, it sends a source selection request to BAShuffler which is housed in the Resource Manager and makes scheduling decisions using the Partially Greedy Source Selection (PGSS) algorithm. The PGSS algorithm is introduced below and the details of the design of BAShuffler can be found in the extended version [10] of this paper.

3.2 Partially Greedy Source Selection

The algorithm of PGSS is presented in Algorithm 1. PGSS assigns every node a *load count*, which indicates how many fetch flows will be created from this node in the immediate or near future. When a shuffle begins, PGSS assumes that there will be a new fetch flow later from every map task. Therefore, it increments the load count of each potential pending source node by the number of map tasks in the node. When PGSS needs to select a source from the pending nodes, it zeros in on the set of source nodes that have the largest remaining load counts ("partial") and selects the one that gives that maximum MMF bandwidth utilization ("greedy").

The advantage of selecting the source from the heaviest loaded nodes is that this incurs a small scheduling overhead. Suppose that the number of nodes in the cluster is N, the number of existing flows in the network is F, the number of pending sources is $M(M \leq N)$, and the number of the heaviest-loaded nodes is $K(K \leq M)$. Given that the time complexity for obtaining the MMF bandwidth utilization of the specific network from a communication pattern is $O(N+F)$, the time complexity of PGSS is $O(K \times (N+F))$ for scheduling each request, instead of $O(M \times (N+F))$ if all the pending sources are considered.



(a) Uneven Pattern (b) Even Pattern

Figure 2: Scenarios of Selecting the Source in Different Flow Patterns

Table 1: MMF Bandwidth Allocation of Uneven Flow Pattern in Homogeneous Network

Selected Flow	a	b	c1	c2	Overall
Nil	3	3	-	-	6
c1	3	3	3	-	9
c2	3	3	-	6	12

Table 2: MMF Bandwidth Allocation of Uneven Flow Pattern in Heterogeneous Network

Selected Flow	a	b	c1	c2	Overall
Nil	6	6	-	-	12
c1	3	6	3	-	12
c2	6	6	-	6	18

3.3 Applying PGSS

We illustrate how PGSS is applied when selecting a source based on the notion of MMF allocation in both homogeneous and heterogeneous network settings, corresponding to the capacities of the access layer links being the same or different, respectively.

Fig. 2 depicts two scenarios of either uneven or even flow pattern, where even means that the numbers of flows into or out of all the nodes are the same, and uneven otherwise. In the homogeneous network setting, the three nodes, A, B, and C, have the same uplink and downlink bandwidth capacities, which are 6, 6 and 6, respectively; whereas in the heterogeneous setting, there capacities are 6, 6 and 12, respectively. The solid arrows represent the existing fetch flows and the dashed arrows represent the new flows that can be selected. Now, a fetcher in Node B becomes available and PGSS needs to decide a source node (A or C) to fetch the data. Assume that both Node A and Node C are the heaviest-loaded nodes.

3.3.1 Homogeneous Network

For the homogeneous network setting, the MMF bandwidth allocation of each flow before or after the selection of a new flow is shown in Table 1, where "Nil" in an entry means before the source selection. Different selection decisions can lead to different MMF bandwidth allocations and overall bandwidth utilizations. PGSS will select Node C as the source, which gives 33% higher overall bandwidth utilization than if Node A is selected. Note that the RSS policy of YARN will have a 50% probability of selecting the source node "A".

3.3.2 Heterogeneous Network

For the heterogeneous network setting, RSS gives rise to an even poorer bandwidth utilization than in the case of a homogeneous network, regardless of whether the flows are evenly allocated across the network or not.

Table 3: MMF Bandwidth Allocation of Even Flow Pattern in Heterogeneous Network

	a1	a2	b1	b2	c1	c2	d1	d2	Overall
Nil	3	3	3	3	3	3	-	-	18
d1	3	3	2	2	2	3	2	-	17
d2	3	3	2	2	4	3	-	2	19

For the uneven flow patterns in Fig. 2(a) with the heterogeneous network setting, the MMF allocation of each flow is shown in Table 2 The overall bandwidth utilization difference between selecting Flow c1 and Flow c2 is amplified in the heterogeneous network (3:2), when compared to the homogeneous network (4:3). PGSS will always select the source (Node C) that brings about the maximum bandwidth utilization.

In the homogeneous network, if the communication pattern of the flows is exactly even, selecting any source node for fetching will make no difference in the overall bandwidth utilization. However, in the heterogeneous network, selecting the right source node can lead to a higher bandwidth utilization.

Fig. 2(b) depicts the scenarios of the even flow pattern, and the capacities of the links follow the heterogeneous network setting. The MMF allocation of the flows before and after selecting the new flows (dashed arrows) is shown in Table 3. Surprisingly but it does happen that the overall MMF bandwidth utilization drops if Flow d1 is selected. PGSS selects Flow d2 to guarantee the maximum bandwidth utilization.

4. EVALUATION

We run BAShuffler in a physical testbed with the heterogeneous network setting. The cluster contains 18 computer nodes, one of which assumes the role of the name node of HDFS, and another one acts as the resource manager of YARN. The remaining 16 nodes are configured as both the data nodes of HDFS and the node managers of YARN. All the 18 nodes are connected to an internal non-blocking switch with GbE ports. To create the heterogeneous network capacities, among the 16 node managers, the bandwidth capacities of the uplinks and downlinks of 8 nodes are manually limited to 160 Mbps, by using the traffic control tool "tc", and the remaining 8 nodes keep to their physical uplink and downlink bandwidth capacity, which is 320 Mbps.

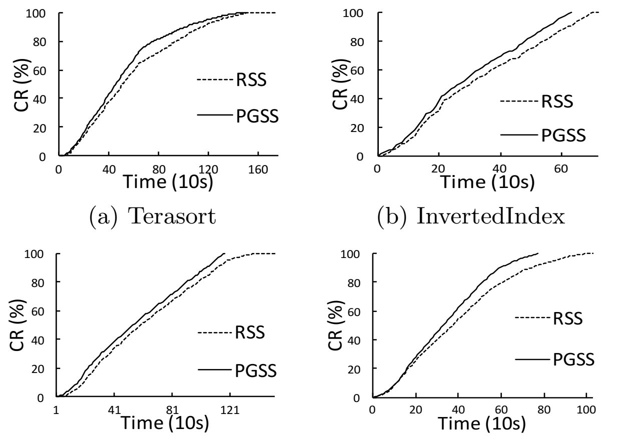
The benchmarks and datasets used are from a realistic MapReduce benchmark suite [1]. We use mainly the shuffle-heavy applications because we want to evaluate the performance of BAShuffler when the shuffle workload can saturate the network most of the time.The sizes of the datasets of the benchmarks are listed in Table 4. Unless specified otherwise, the number of fetchers in each shuffle task is 5 (the default value).

4.1 Shuffle Throughput

The metric of the overall shuffle throughput reflects the cluster's overall bandwidth utilization along the time axis. The overall shuffle throughput is depicted as the cumulative completion ratio of the overall shuffle workload. Fig. 3 shows the cumulative completion ratios of RSS and PGSS in various benchmarks, where PGSS clearly outperforms RSS in all the benchmarks. The overall shuffle throughput improvement is the result of maximizing the overall bandwidth

Table 4: Benchmark Dataset Size (GB)

Benchmark	Input	Shuffle	Output
Terasort	190	190	190
InvertedIndex	200	42	34
SequenceCount	300	180	150
RankedInvertedIndex	150	175	153



(a) Terasort (b) InvertedIndex (c) SequenceCount (d) RankedInvertedIndex

Figure 3: Cumulative Completion Ratio (CR) of the Overall Shuffle Workload of RSS and PGSS in Various Benchmarks along the Time

utilization, e.g., the shuffle throughput speedup due to PGSS is about 12% in the Terasort benchmark.

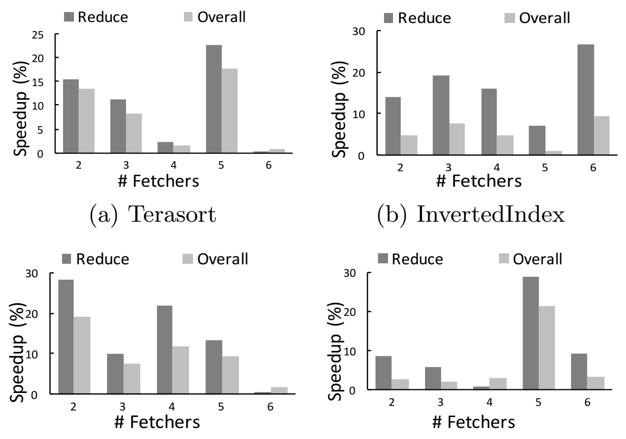
4.2 Completion Time

The reduce completion time is the duration from the time when all the map tasks have finished to the time when the job finishes. Fig. 4 depicts the reduce completion time speedup and the job overall completion time speedup by PGSS as compared to RSS with different numbers of fetchers in each shuffle task. Different numbers of fetchers will create different degrees of traffic congestion in the network. As the benchmarks are reduce-heavy, where the shuffle phase can occupy a major portion of the overall workload, in most cases, BAShuffler not only improves the reduce phase, but also the overall completion time of the jobs rather decisively. For example, in the RankedInvertedIndex benchmark with 5 fetchers, PGSS shortens the reduce completion time by 29% and the overall completion time by 21%. In some cases, the speedup of PGSS is not obvious (e.g., Terasort with 6 fetchers in Fig. 4(a)) which is because the reduce completion time of RSS is already the minimum among all the fetcher settings.

5. CONCLUSION

In this paper, we describe BAShuffler which we implement in YARN to improve the shuffle performance. It schedules the source nodes of the shuffle flows at the application level in order to maximize the overall max-min fairness bandwidth utilization. BAShuffler can significantly increase the shuffle performance especially when the network is heterogeneous in the capacities of its links.

Acknowledgement This work is supported in part by a Hong Kong RGC CRF grant (C7036-15G).



(a) Terasort (b) InvertedIndex (c) SequenceCount (d) RankedInvertedIndex

Figure 4: Reduce Completion Time Speedup and Job Overall Completion time Speedupof PGSS

6. REFERENCES

[1] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012.

[2] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. *SIGCOMM*, 2013.

[4] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, 2011.

[6] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *SIGCOMM*, 2014.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[8] 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.

[9] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*, 1988.

[10] F. Liang and F. C. M. Lau. Bashuffler: Maximizing network bandwidth utilization in the shuffle of yarn. http://i.cs.hku.hk/%7Efliang/paper/BAShuffler.pdf.

[11] R. Niranjan 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*, 2009.

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

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *SOCC*, 2013.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.