Maximizing Shuffle Network Bandwidth Utilization by Application-Level Flow Scheduling

Feng Liang, *Member, IEEE*, Francis C. M. Lau, *Senior Member, IEEE*, Heming Cui, *Member, IEEE*, and Cho-Li Wang, *Member, IEEE*

Abstract—The shuffle operation is pervasive in distributed applications and consists of many network flows across computer nodes in datacenters. ~~The shuffle operation, which relies on the network bandwidth to transfer data across computer nodes in the cluster, is pervasive in the distributed applications. However, unlike the node-monopolistic resource such as memories and CPU cores, the existing distributed resource management platforms does not offer a fine-grained resource allocation mechanism for the network bandwidth.~~ The problem of underutilization of the network bandwidth can greatly affect the completion time of shuffle tasks having large volumes of data to transfer within the cluster, the shuffle-heavy MapReduce jobs being a typical one. Unfortunately, the impact of bandwidth underutilization is more pronounced if the network bandwidth capacities of the nodes ~~in the cluster~~ are varied. ~~This heterogeneous network bandwidth environment widely exists in commodity clusters and enterprise data centers during upgrading periods.~~

In this paper, we demonstrate how ~~the communication flows with randomly selected source nodes in the shuffle phase cannot fully utilize~~ shuffle flows can significantly under-utilize the network bandwidth capacity in a max-min fair network. We present BAShuffler, a bandwidth-aware shuffle scheduler, that ~~can maximize~~ maximizes the overall network bandwidth utilization ~~by using a greedy or a partially greedy method to schedule the source nodes of the shuffle flows~~ in shuffle operations. ~~Instead of working on the network level that schedules the bandwidth of flows or the routing of data packages, BAShuffler uses the application-level approach that schedules flow source nodes based on the precise bandwidth allocation estimation without changing the underlying network.~~ Compared to the existing network-level solutions, BAShuffler takes the first step to adopt the application-level approach that schedules flow source nodes based on the bandwidth allocation estimation without changing the underlying network. Besides simulations, we evaluate BAShuffler with a variety of realistic benchmarks in testbeds comprising a physical cluster and an EC2 virtual cluster respectively. ~~Simulation is also used to evaluate the performance of BAShuffler in clusters of different scales.~~ Experiment results show that BAShuffler can significantly improve the cluster shuffle throughput and reduce the execution time of shuffle tasks by up to 29% as compared to the original Hadoop, especially in ~~the~~ heterogeneous networks. ~~bandwidth environment.~~

Index Terms—YARN, MapReduce, Shuffle, Network Scheduling.

1 Introduction

The shuffle operation is pervasive in the MapReduce-like [1] distributed operations, such as the "*join*" operation in the distributed database systems [2], [3], [4], the "*reduce*" tasks in the MapReduce systems [1], [5] and the "*aggregateByKey*" operation in Spark [6]. The shuffle operation may require a large amount of network bandwidth and is observed to dominate the job completion time in the datacenter. A study on the Yahoo! datacenter work trace suggests that as many as 70% of the jobs are shuffle-heavy and other studies show that the shuffle completion time can account for 33% of the overall completion time. Optimizing the shuffle performance is critical for these shuffle-heavy jobs.

Many popular distributed resource management platforms such as YARN [5] (the latest Hadoop [7]) and Mesos [8] are developed to cope with the MapReduce-like applications with shuffle operations. By encapsulating the computing resources (usually memories and CPU cores) of the cluster via the abstraction of containers, the distributed platforms exercises a fine-grained control over the resources. If the performance of the distributed tasks is mainly decided by the memories and the CPU cores, which is true of many compute-intensive jobs, YARN can do very well in terms of resource utilization, assuming that the users are able to properly specify the resource sizes for the tasks, either by experience or by estimation. In reality, however, the shuffle-heavy tasks need to transfer non-trivial amounts of data via the local area network among the tasks. The performance of these communicative tasks can be largely affected by the network bandwidth and the scheduling of the communication flows. Disk I/O, on the other hand, has less an effect on such communication time [9].

Unfortunately, the throughput of a communication flow is mainly determined by the available network bandwidth of the connection on both endpoints of the source and the destination. The network is the shared resource of a pair of nodes even for one single task that transfers data between them. For this reason, unlike memories and CPU cores, the network bandwidth cannot simply be encapsulated in resource containers that stand for the isolation of the monopolistic resources.

In the distributed platforms, the shuffle tasks by default would try to evenly distribute the workload of the network by randomly selecting the source to fetch the map output data. If the links connecting all the nodes in the cluster

- *F. Liang, F.C.M. Lau, H. Cui and C.-L. Wang are with Department of Computer Science, The University of Hong Kong.*
  E-mail: F. Liang- loengf@connect.hku.hk, F.C.M. Lau- fcmlau@cs.hku.hk, H. Cui- heming@cs.hku.hk, C.-L. Wang- clwang@cs.hku.hk

are more or less equal in terms of bandwidth and if the number of network connections is large, this random source selection policy could prevent some nodes becoming a bottleneck. Obviously, however, without monitoring the actual connection allocations in the cluster, this simple random source scheduling approach cannot offer any bandwidth guarantee. And it will likely lead to suboptimal performance if the network is a *heterogeneous* one, i.e., the network bandwidth capacities of the links are different. The reason will be analyzed with examples in Section 3.4. A heterogeneous network environment can be quite common in practice for distributed platform users. For example, Hadoop is designed as the general distributed processing platform for clusters comprising off-the-shelf machines [1]. The physical Hadoop clusters could be fit with a variety of servers and network devices of different configurations [10]. Even in enterprise data centers, heterogeneity exists during the long period of devices upgrading [11]. ~~In a virtual cluster, EC2 virtual machines could be running at different network speeds.~~

Our key insight is that the application-level scheduling of the source nodes of the shuffle flows can leverage the sufficient shuffle information to maximize the network bandwidth utilization. ~~The shuffle information includes the shuffle-related information (e.g., the arriving patterns and the source/destination pairs of the flows) and the network-level information (e.g., the bandwidth capacity of the network links).~~ ~~A central scheduler in the distributed resource management platform, e.g., the Resource Manager of YARN, makes use of the shuffle information and schedules the flow source nodes for all the MapReduce-like programs running on this platform.~~ Although some work focuses on the network level [12], [13], [14], [15], the well-defined network level model cannot fully capture the the behavior of the shuffle operation, e.g., the dynamic and casual arriving time of the fetch flows when a map task finishes. A central scheduler at the application level has the potential to achieve the scheduling results based on both the dynamic shuffle-oriented information (e.g., the arriving patterns and the source/destination pairs of the flows) and the network-level information (e.g., the bandwidth capacity of the network links). Moreover, the application-level shuffle scheduler enables the distributed platforms to work on unmodified operating systems and general network devices. We will discuss in detail the rationality of improving the shuffle performance by scheduling the sources of the fetch flows at the application level in Section 2.2.

In this paper, we present BAShuffler, a network-bandwidth-aware shuffle scheduler, that can maximize the network bandwidth utilization of the shuffle operation without changing the underlying network and the existing MapReduce-like interfaces. BAShuffler applies the partially greedy source selection (PGSS) algorithm to select the appropriate source nodes of the TCP fetch flow so that the the network bandwidth utilization is maximized. We implement BAShuffler based on YARN and apply it in several benchmarks. We conduct experiments on both a physical cluster and an EC2 virtual cluster to evaluate the real performance of BAShuffler. Experiment results show that BAShuffler can significantly increase the shuffle throughput and reduce the total job completion time, by up to 29% for shuffle-heavy jobs in the physical and the virtual clusters as compared to the original YARN. We also use simulations to mimic the clusters of different scales. The simulation results show that comparing to the random-source-selection method adopted by YARN, PGSS can improve the cluster network bandwidth utilization especially in a heterogeneous network environment, without low scheduling overhead.

The rest of the paper is organized as follows. In Section 2, we provide some background information on the shuffle mechanism and the max-min fairness behavior of the TCP communication, and discuss the motivation of improving the shuffle performance by scheduling at the application level. The design and the implementation details of BAShuffler and PGSS are given in Section 3. Section 4 presents the evaluation of BAShuffler. We discuss the related works in Section 5 and conclude the paper in Section 6.

2 Background, Motivation and Discussion

In this section, we give an overview of the resource management mechanism and the shuffle policy of YARN and discuss the drawbacks of the current design. We discuss the motivation of scheduling the shuffle flows at the application level and introduce the max-min fairness behavior in TCP communication, which is a convenient bandwidth allocation estimation tool used by BAShuffler.

2.1 Resource Management and Shuffle in YARN

YARN manages the computing resources in the cluster as a collection of containers, each of which consists of memories and CPU cores. The jobs are sub-divided into tasks according to a certain specific computing paradigm, which are then assigned to their respective allocated containers having the required amounts of memories and CPU cores.

The resource management mechanism of YARN requires the users to estimate the amounts of memories and CPU cores that the tasks will use. An underestimation would result in a container being too small for the task to run, while too large a container may deprive other tasks of their resources and may even prevent them from running. Either way will impact the cluster's overall performance. Although much work [16], [17] has been done to optimize the compute-intensive resource utilization (of CPU, memory, etc.) in Hadoop, there has not been any feasible solution for controlling the network bandwidth in the cluster. Indeed, the memory and the CPU core abstraction of YARN cannot properly model the performance of tasks that are network-centric such as the shuffle phase of MapReduce.

MapReduce is the most common distributed computing paradigm in YARN, especially when performing big data analysis. MapReduce can be divided into two interfaces: map and reduce. The map interface processes the input data and collects a list of values corresponding to different keys; the reduce interface takes the output of the map primitive as input and generates a value for each key. Both interfaces run as a set of map or reduce tasks in the YARN containers in the cluster.

A reduce task, before the actual reduce semantics execute, needs to fetch the map output data from different partitions on different nodes for the keys/values to reduce, which is called a "**shuffle**". Each fetch translates to an HTTP request to the server to transfer the map output of a specific partition on a node to the reduce node, which can simply be conceived as a TCP flow. There are usually many fetches from different source nodes in a shuffle. A shuffle is a many-to-one TCP communication instance, and all the shuffles together for a MapReduce job make up a many-to-many TCP communication. When a shuffle begins, it starts off a specific number of threads, called fetchers, each of which then randomly selects a pending fetch from the set of all the fetches of the shuffle. In other words, a specific number of fetchers will randomly select the source nodes to transfer the map output data in parallel. This random source node selection (RSS) approach adopted by YARN can approximately evenly distribute the network transfer workload over different source nodes, and reduce the probability of the traffic congesting in the uplink of a few source nodes.

RSS is widely used in the popular distributed platforms as the major shuffle source selection algorithm [5], [6]. However, as we have commented, RSS cannot fully utilize the network bandwidth when the network bandwidth capacities of the network links connecting the nodes are different, as in a heterogeneous network environment. In such an environment, some links might become congested by too many flows, while some others leave much of their network bandwidth unused. A network bandwidth shuffle scheduler that can select that proper source node to fetch data from is necessary in order to fully utilize the overall network bandwidth available in the cluster.

2.2 Application-Level Shuffle Scheduling

To improve the shuffle performance, a solution can be devised to operate at the network level or the application level, with both pros and cons at either level.

At the network level, ideas such as performance isolation [18] and fair sharing of network resources [12], [19] 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 of individual flows may not always lead to improved shuffle performance.

Therefore, it is better that the network-level optimization can consider the flows belonging to one shuffle as a whole when scheduling. The "coflow" model [14], [15] has been proposed to allow scheduling the network based on the granule of a collection of application-level correlated flows. But nevertheless, neither the coflow nor any other pure network-level model can actually describe the runtime status of the shuffle phase due to information isolation between the network level and the application level. When there is a large set of map outputs that need to be fetched during the shuffle, as the application level can only create a limited number of fetch flows at one time (5 per reduce task by default) due to system limits, the remaining map outputs will be left pending until there are available fetch workers later. The network level (or coflow here) is only aware of the existence of the flows created, but not the potential flows of the same shuffle pending at the application level. With this limitation, the shuffle operation cannot exactly fit into the coflow model. Minimizing the completion time of the coflow is not the same thing as minimizing the completion time of the shuffle operation.

The shuffle completion time is the execution time of the shuffle operation from the time the first fetch flow is ready to start to the time the last fetch flow finishes. 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 destinations) and the network level information (the network fabric, the routing, the 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 communication between the application level and the network level could be prohibitive. What is more important is that such a cross-layer approach would violate the principle of the isolation of the application level and the network level [20].

Application-level shuffle scheduling has the advantage of knowing the true runtime status of the shuffle. "Application-level" is the term relative to the network layers in the OSI model, but is not limited to the distributed application (e.g., MapReduce programs). ~~The distributed resource management platforms like Mesos and Hadoop is also working at the application level.~~ Although it does not make efforts to improve 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 MMF in the TCP network) to obtain the near-optimal solution.

The benefits of application-level shuffle scheduling are: 1) Besides bandwidth capacities of the links, it can make use of the dynamic application-level shuffle information, including the pattern of the arriving time of the flows and their sources and destinations; 2) The distributed platforms can work on unmodified operating systems and general network devices.

2.3 Max-Min Fairness in TCP Communication

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 [21]. The MMF of TCP has been extensively analyzed and verified in the literature [22], [23], [24], [25]. Although it cannot accurately model the exact behavior of the TCP communication, the MMF model is acceptable and appropriate for approximating the network behavior, and can lead to useful conclusions in various application settings.

When there is MMF in the communication network, there exists a feasible and unique bandwidth allocation such that an attempt to increase the allocation of any flow will be at the cost of decreasing the allocation of some other flows with equal or smaller allocation. Each data flow must have a *bottleneck link*, which is saturated, and of all the data flows sharing the bottleneck link, this data flow occupies the overall maximum bandwidth.

The MMF allocation can be derived by a progressive filling algorithm if the bandwidth capacities of the links and the routing of all the data flows are known in advance (as apposed to best-effort networks). The first step is to find out the saturated links. Suppose that all data flows passing through a link will get equal bandwidth shares, i.e., the bandwidth capacity of the link divided by the number of flows going through it. The links with the least average bandwidth share are the saturated links, and this least bandwidth share is the final bandwidth allocated to these flows on the saturated links. The second step is to update the set of links pending for allocation and available bandwidth capacities of the links. The saturated links and the all flows on them are removed; for all the other links that are on the paths of the removed flows, the bandwidths occupied by the removed flows are subtracted from the link bandwidth capacities. Finally, the first and second steps are repeated given the information on the updated link bandwidth capacities and the remaining data flows, until all data flows have been allocated bandwidth.

By using this method, the current bandwidth allocated to each flow and the utilization of the overall bandwidth can be estimated in TCP communication that behaves in the MMF way, 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 [18], [26], [27], it is practical to simplify the datacenter fabric as a non-blocking switch [13], [14], [15], [28], [29], [30]. In this case, the bottleneck links of the flows lie in the access layer, which directly connect to the nodes. When figuring out the MMF allocation of the TCP flows, the network topology and the paths of the data flows can be ignored. By merely knowing the bandwidth capacities of the uplinks and downlinks in the access layer and the source and destination nodes of the TCP flows, we can obtain the bandwidth allocation of the flows easily. The non-blocking switch abstraction largely simplifies the estimation of the MMF bandwidth allocation.

In the rest of the paper, we assume that the bottleneck links of the TCP links are all in the access layer, and optimize the TCP flow scheduling in the shuffle without the knowledge of the network topology of the cluster.

3 BASHUFFLER

In this section, we introduce the design of our bandwidth-aware shuffle scheduler, named BAShuffler. BAShuffler improves the throughput of the shuffle phase of MapReduce jobs in YARN by selecting the appropriate source node for fetching map output data from in order to increase the network bandwidth utilization. The greedy source selection method and the partially greedy source selection method are applied for selecting the source nodes. BAShuffler optimizes the network transfer throughput at the application level, without having to modify the underlying network protocols. It will not affect the performance of the datacenter networks when there are other non-MapReduce jobs running. BAShuffler schedules the sources of the fetches in the background without any change to the current interfaces of MapReduce, and users need not be aware when designing the logic of their jobs. BAShuffler can be implemented seamlessly in YARN and existing MapReduce jobs can run in YARN without changing any code in the presence of BAShuffler.

3.1 Design of BAShuffler

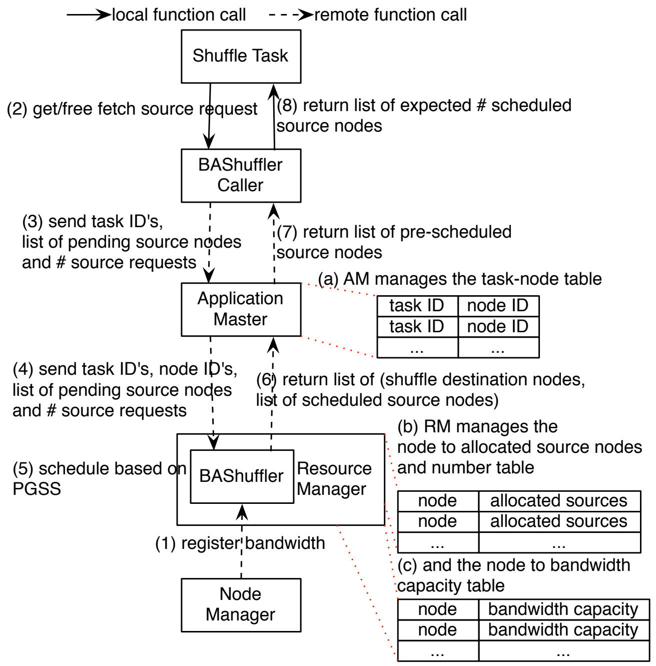
3.1.1 Design Considerations

When improving the shuffle performance, it is not wise to sacrifice the execution time of the other phases of the jobs. BAShuffler should not negatively impact the resource allocation for the other map tasks and reduce tasks. Also BAShuffler prefers not to change the task assignment policy of the existing system as changing it will cause chaoses in the issues of task locality and task dependency. The only effect of BAShuffler shall be a faster shuffle operation. With the other phases unaffected, the job execution time is improved with faster shuffle.

In order to improve the shuffle performance, it is necessary for the system to schedule network bandwidth as a kind of resource. However, it is not appropriate to encapsulate network bandwidth in a container like what is done for memory or CPU cores, for the following two reasons: a) the network bandwidth occupied by the communication flow is decided by the two nodes on both ends of the flow and cannot be represented in a container that resides on a single node; b) the task in a container may only use the network bandwidth as the necessary resource for some period of time throughout the whole lifecycle of the container, and a container for network bandwidth cannot properly model the resources a task requires for its execution. Therefore, BAShuffler will not change the allocation mechanism of resource containers for the tasks. Instead, BAShuffler focuses on the shuffle phase of the reduce task, which is already running in an allocated container.

During the shuffle phase, there are many factors that can affect the throughput or the execution time of the shuffles. For example, the CPU time slices for source nodes to send the map outputs via the fetch flow and the straggler of a map task that prevents the shuffle operation having all the map outputs ready. These factors are related to the dynamic environment of the distributed management platform. Some of them can be avoided by a better resource isolation mechanism. For instance, a good resource isolation mechanism should make sure the CPU time slices guaranteed for sending the flow data will be stolen by other compute-intensive jobs. BAShuffler does not consider these factors.

One factor we want to discuss here is the number of fetchers of each shuffle fetching the data concurrently. This number directly decides the traffic workload in the cluster. Too many fetchers can jam the network (the TCP incast problem [31]), while too few fetchers may increase the probability of the network links being underutilized. The optimal number of fetchers in fact varies with the network settings as well as the application, but generally need not be very large (as suggested by the incast problem). According to some of the Hadoop best practices (e.g. [7]), tuning the number of fetchers is not a major performance consideration. Considering there can be multiple task containers in one node, we consider 2 to 6 fetchers per shuffle task would be acceptable. For the experiments that evaluate the other factors than the fetcher number, such as different jobs


local function call
remote function call
Shuffle Task
(2) get/free fetch source request
(8) return list of expected # scheduled source nodes
BAShuffler Caller
(3) send task ID's, list of pending source nodes and # source requests
(7) return list of pre-scheduled source nodes
(a) AM manages the task-node table
Application Master
(4) send task ID's, node ID's, list of pending source nodes and # source requests
(6) return list of (shuffle destination nodes, list of scheduled source nodes)
(b) RM manages the node to allocated source nodes and number table
(5) schedule based on PGSS
BAShuffler
Resource Manager
(1) register bandwidth
(c) and the node to bandwidth capacity table
Node Manager

task ID	node ID
task ID	node ID
...	...

node	allocated sources
node	allocated sources
...	...

node	bandwidth capacity
node	bandwidth capacity
...	...

Fig. 1. Architecture Design of BAShuffler

and input sizes, we do not bother with the optimal fetcher number and simply used Hadoop's default number, which is 5.

Another important factor is the communication pattern of the fetch flows. The communication pattern mainly refers to the mapping from the the flow sources to the flow destinations. Different communication patterns can lead to different bandwidth utilization status, which we will illustrate in the coming sections. BAShuffler focuses on this factor and schedules the communication pattern. To schedule the communication pattern that can yield maximum bandwidth utilization, BAShuffler selects the source node of the fetch based on the MMF bandwidth estimation when a new fetcher is free.

3.1.2 Architecture of BAShuffler

The logical position where the BAShuffler components are embedded in YARN is shown in Fig. 1. The solid arrow lines represent local signals or data communications between components by method calls in the same node, while the dashed arrow lines represent remote procedure calls (RPC) between components in different nodes.

At startup, the node managers register the bandwidth capacities of the uplink and downlink of the node with BAShuffler. The users can obtain the capacities of these access layer links of the nodes either from the specification of the cluster configuration or by a simple bandwidth evaluation tool.

When a shuffle task needs to schedule a new fetch, with the machine executing the shuffle task as the destination node, it asks the BAShuffler caller to initiate a request to the remote BAShuffler to decide the source node to schedule, from a set of available nodes that has finished some map task. The BAShuffler caller sends the necessary shuffle information to the corresponding application master, including the shuffle task ID, the list of pending nodes to fetch the data from and the request number of source nodes. When the application master receives the shuffle scheduling request, it caches the request and extracts the node information from the job information maintained in the application master. In the heartbeat from the application master to the resource manager, besides the original container allocation requests of YARN, the application master also sends the shuffle source scheduling request to BAShuffler, which lies in the resource manager. BAShuffler applies PGSS to schedule the fetch source of the shuffle tasks, which will be introduced in the following sub-sections. BAShuffler adopts the centralized scheduling policy and make the source scheduling decision based on the link bandwidth capacities and scheduling status of the whole cluster, e.g., the records of the source and destination nodes of the scheduled flows, with the fact that this information is easily at hand of the resource manager. When the source nodes for fetching are selected, BAShuffler will update the current scheduling information and the result is transferred back to the shuffle task in the reversed path.

When the shuffle task finishes fetching the data from a source and wants to free the flow, it will use the BAShuffler caller to update the source scheduling information in BAShuffler along the same path.

When a node manager is added or deleted from the cluster, the node manager registers or deregisters its link capacities information (and also the existing flow information when deregistering) to BAShuffler, just as at startup. Adding or deleting a node in the cluster will not add additional workloads of the users installing and configuring the system.

Algorithm 1: Partially Greedy Source Selection

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

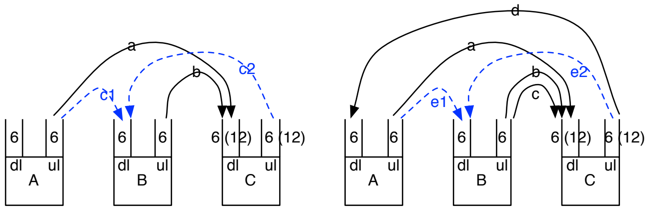
Fig. 2. Pseudo Code of the Partially Greedy Source Selection Algorithm

3.2 Greedy Source Selection

A straightforward method to improve the shuffle throughput when selecting the source is to use the greedy method for online scheduling: whenever there is a new fetch to schedule, the shuffle selects the pending source node such

TABLE 1
Time Complexity of Different Scheduling Algorithms for Scheduling Each Request

RSS	**GSS**	**PGSS**	**Online Optimal**
$O(1)$	$O(M \cdot (N+F))$	$O(K \cdot (N+F))$	$O(\frac{(P+M-1)!}{P \cdot P! \cdot (M-1)!} \cdot (N+F))$



(a) Sparse Case

(b) Intensive Case

Fig. 3. Scenarios of Selecting the Source in Uneven Flow Pattern

that after selecting it, the MMF bandwidth of the communication pattern is no less than that when any other pending source node is selected. We call this shuffle scheduling policy Greedy Source Selection, or GSS.

GSS is an effective method as long as it is not the worst case: the selections of the shuffles focus on a part of the pool of the nodes too much so that later on, the selection of the remaining source nodes of all the shuffles can only be concentrated in the other part of the pool. In this case, in the latter period of source scheduling, the bandwidth of some nodes cannot be used at all as all the map outputs have already been fetched. The total bandwidth utilization might fall dramatically in this worst case.

3.3 Partially Greedy Source Selection

Just like the other online scheduling algorithms [32], [33], GSS does not predict the arrival pattern of the coming source nodes. It makes sure that the maximum transfer throughput of the whole cluster is achieved during the short period following the selection of each source, but it cannot guarantee the maximum average throughput of the cluster for a much longer period of time.

A better shuffle source scheduling algorithm is possible if we can predict the arrival pattern of the source nodes. There is however almost no way to know the arrival pattern of the shuffle tasks in advance as when and where the shuffle task starts depends on the allocation of the reduce containers, which is a stochastic process. Nevertheless, after a shuffle has started, the future pairs of the source and destination of the fetch flows can be approximately predicted. The destination of the fetch flows is the node where the shuffle task executes, and the sources are all the nodes where there are input data for the map tasks of the same job (we ignore the case when there happens to be no map output partition for the shuffle, which is almost impossible for reasonably large input datasets).

Therefore, we can improve GSS to avoid concentration via what we call Partially Greedy Source Selection (PGSS), whose algorithm is shown in Fig. 2. PGSS marks all the nodes with a *load count*, which indicates how many fetch flows will be created from the corresponding source nodes in the immediate or near future. When a shuffle begins, PGSS predicts 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 first ("partial") and selects the one that gives that maximum MMF bandwidth utilization ("greedy").

PGSS may not increase the overall bandwidth utilization as compared to GSS, as PGSS only selects the source nodes from the heaviest-loaded sources, which are a subset of the domain of GSS. However, PGSS can avoid the concentration of selection from a small set of sources, while maintaining the competitive bandwidth utilization, especially in the long run. Another advantage of PGSS against GSS is that PGSS may incur even a smaller scheduling overhead, as PGSS only selects the source nodes from the heaviest-loaded nodes.

Although GSS and PGSS are not the online optimal scheduling solutions for the shuffle source selection problem, they are much simpler in terms of time complexity than the online optimal solution. Suppose that the number of nodes in the cluster is N, and during the shuffle, at a specific scheduling moment, the number of existing flows in the network is F. P fetchers are requesting from the same set of pending sources of size M at the same time, while the number of the heaviest-loaded nodes is $K (K \leq M)$. The online optimal solution considers every P-combination with M repeated source nodes and then choose the combination that obtains the maximum bandwidth utilization. Given that the time complexity for obtaining the MMF bandwidth utilization of a specific network from a communication pattern is $O(N + F)$, the time complexities of different shuffle source scheduling algorithms for scheduling a fetcher are summarized in Table 1. RSS, by random selection, gives the least time complexity. The time complexities of GSS and PGSS are much smaller than the online optimal algorithm, where generally, PGSS suffers even less scheduling overhead than GSS. It is verified in Section 4 that the scheduling performance of PGSS is close to the online optimal.

Currently, GSS and PGSS do not differentiate the fetch flows belonging to different shuffle tasks or different jobs during scheduling. It schedules each fetch to maximize the MMF utilization for the whole cluster, but not just for the jobs that the fetch belongs to. Also, GSS and PGSS do not consider the workload size of the flows (the data size to transfer in the flow), as the bandwidth utilization efficiency, instead of the flow completion time, is the major concern of BAShuffler. In fact, the shuffle fetch flows are not batched tasks as some fetch flows are unknown yet and the future flow tasks can come dynamically at casual time. It is unpractical to consider the shuffle completion time as the performance goal for the online flow tasks. The goal of maximizing bandwidth utilization is reasonable in the shared environment, e.g., in multi-tenant clusters, and we believe that higher overall bandwidth utilization will lead to shorter completion time.

TABLE 2
Max-Min Fairness Bandwidth Allocation of Flows of the Sparse Case in the Homogeneous Network

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

TABLE 3
Max-Min Fairness Bandwidth Allocation of Flows of the Intensive Case in the Homogeneous Network

Selected Flow	a	b	c	d	e1	e2	Overall
Nil	2	2	2	6	-	-	12
e1	2	2	2	6	4	-	16
e2	2	2	2	3	-	3	12

TABLE 4
Max-Min Fairness Bandwidth Allocation of Flows of Uneven Flow Pattern in Heterogeneous Network Setting

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

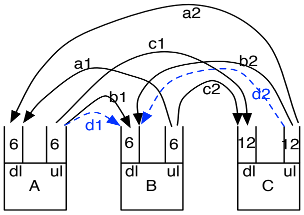


Fig. 4. Selecting the Source in the Even Flow Pattern

3.4 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. The analysis can easily extend to a large scale of nodes and flows in the datacenter, as the MMF behavior follows the same method of estimation.

As dicussed in Section 2.3, we assume that the bottleneck links of the TCP flows are the access links to the computer nodes. We consider both the homogeneous and the heterogeneous network settings, which are distinguished by whether the capacities of the access layer links are the same or vary. Although RSS can distribute the flows more or less evenly, it cannot guarantee exactly the same workload (or number of flows) on each node.

Fig. 3 depicts two scenarios of the uneven flow pattern, where uneven means that the numbers of flows into or out of all the nodes are not the same, and even otherwise. In the sparse case, most of the links are idle, while in the intensive case, the links are almost saturated by the flows. 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 newly coming 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.

3.4.1 Homogeneous Network

In the homogeneous network setting, the MMF bandwidth allocation of each flow before or after selecting a new flow is shown in Table 2 (for the sparse case) and Table 3 (for the intensive case), respectively, where “Nil” in an entry means before the source selection. Different selection decisions can lead to different flow bandwidth allocations and overall bandwidth utilizations. In the sparse case (Fig. 3), 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 Node A, not guaranteeing the maximum utilization of the bandwidth even in the homogeneous network setting.

In the intensive case (Fig. 3), if we have the assumption that the upload (down) rates of the flows are only decided by the uplink (downlink) capacity and the number of the flows sharing that link as in [13], it will make no difference whether to choose Flow e1 or Flow e2, as both source nodes have one outgoing flow from them (Flow a and Flow d). However, by the notion of MMF, PGSS will select e1 that can maximize the overall bandwidth utilization. Applying MMF to estimate the bandwidth allocation can help make better decisions when selecting the source nodes at the application level.

3.4.2 Heterogeneous Network

In the heterogeneous network setting, the randomly source selection policy can have an even greater (negative) impact on the bandwidth utilization than in a homogeneous network, no matter whether the flows are evenly allocated across the network or not.

Look at the uneven flow patterns in Fig. 3 again. In the heterogeneous network setting, the MMF bandwidth allocation of each flow is shown in Table 4 The overall bandwidth utilization difference between selecting Flow c1 and Flow c2 is amplified in the heterogeneous network (3:2), 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, that is where every link has the same number of flows, selecting any source node for fetching will make almost 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. 4 depicts the scenarios of the even flow pattern, and the capacities of the links follow the heterogeneous setting. The MMF allocation of the flows before and after selecting the new dashed flows is shown in Table 5. 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.

TABLE 5
Max-Min Fairness Bandwidth Allocation of Flows of Even Flow Pattern in Heterogeneous Network Setting

	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

Overall, neither RSS nor selecting the source based on the number of flows on the link can help make the right decision that can maximize the bandwidth utilization in the MMF network. However, by using the knowledge of the bandwidth capacities of the access layer links and the TCP flow communication patterns (sources and destinations of the flows), PGSS can easily find out the proper source node for achieving the maximum transfer throughput of the flows by the behavior of MMF.

3.5 Implementation Details

Rearranging Selecting Sequence: PGSS applies the "first-hit" approach when selecting the source node that achieves the maximum bandwidth utilization. For a specific sequence of the pending source nodes, PGSS will always prefer the source node at the front of the sequence among the source nodes with the same maximum bandwidth utilization if they are selected. To avoid the case where the source nodes in the front of the sequence starve the nodes in the tail, PGSS rearranges the sequence of the pending source nodes randomly before it starts to find the first-hit source node. This guarantees that every source node that can achieve the maximum bandwidth has the chance to be selected, and reduces the probability of concentration during the latter period of scheduling.

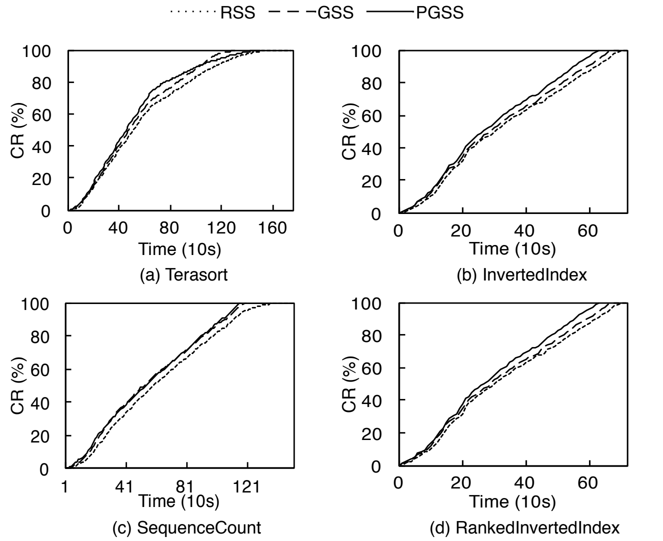
Pre-Scheduling: Usually, the shuffle task will have only one free fetcher at a time, and the shuffle needs to ask the BAShuffler caller to request a scheduled source every time (Fig. 1). For every source request from the BAShuffler caller to the BAShuffler, it takes two segments of remote call time, one segment of heartbeat interval time and two segments of thread scheduling time. In fact, the pending sources for fetching are known before the free fetcher asks for the source. The BAShuffler uses the *pre-scheduling* strategy, which requests a few of the scheduled sources from the BAShuffler at a time, and when the shuffle task needs a source to fetch, the BAShuffler caller can return one source to the shuffle immediately. Although the actual flow pattern may lag slightly behind the scheduling pattern stored in the BAShuffler, the pattern will soon catch up when the scheduled sources in the cache of the BAShuffler caller are consumed.

4 EVALUATION

To evaluate the performance of BAShuffler, we conduct experiments in a physical cluster of heterogeneous network setting with realistic benchmarks and datasets. The performance improvement of BAShuffler will also be verified in an Amazon EC2 virtual cluster, whose underlying network setting is unknown to the users. Furthermore, we also evaluate the performance of BAShuffler in clusters of different scales and degrees of heterogeneity by simulation.

TABLE 6
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

Fig. 5. Cumulative Completion Ratio (CR) of the Overall Shuffle Workload of RSS, GSS and PGSS in Various Benchmarks along the Time in the Physical Cluster

4.1 Physical Testbed

To verify the benefits of BAShuffler in scheduling the shuffle fetch flows with awareness of the network bandwidth allocation behavior, we run BAShuffler in a physical testbed, the Gideon-II cluster [34] in the heterogeneous network configuration.

The physical cluster contains 18 computer nodes, where one node assumes the role of the name node of HDFS, and one node 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. Each node is equipped with 2 quad-core, 32 GB DDR3 memory and 2 × 300 GB SAS hard disks running RAID-1. All the 18 nodes run with Scientific Linux 5.3 and are connected to an internal non-blocking switch with GbE ports. To create the heterogeneous network capacity, among 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 [35]. 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, and ignore the other applications whose shuffle workloads are quite low. The sizes of the datasets of the benchmarks are listed in Table 6.

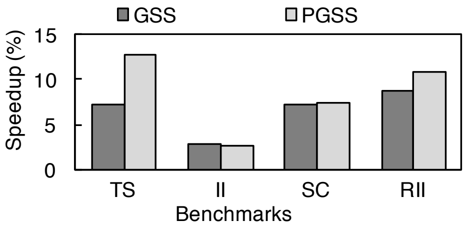


Fig. 6. GSS and PGSS Speedup of Average Shuffle Rate of Each Shuffle Task in comparison to RSS in Benchmarks Terasort (TS), InvertedIndex(II), SequenceCount (SC) and RankedInvertedIndex (RII) in the Physical Cluster

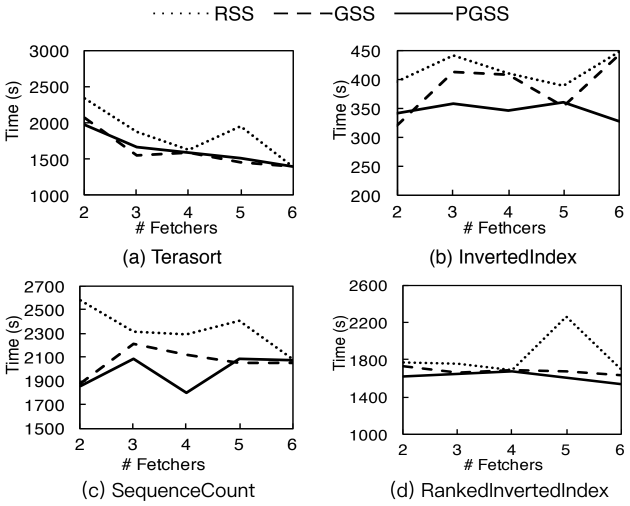
BAShuffler is configured in advance to have the knowledge of the link bandwidth capacity of each node. Unless specified otherwise, in both the physical testbed and the virtual testbed introduced later, the number of fetchers in each shuffle task is 5 (the default value), and the number of reduce tasks of each job is configured to be the total number of node managers in the cluster. We compare GSS and PGSS with RSS, and RSS is the default and the only presently known application-level shuffle scheduler in YARN. The following subsections report the overall shuffle throughput of the cluster, the shuffle rate of each shuffle task, the reduce phase completion time and the job overall completion time. We also measure the performance of BAShuffler with input data of different sizes and the performance in the shared environment with multiple concurrent jobs. But the results will be introduced later together with the results of the virtual cluster.

4.1.1 Shuffle Throughput

The throughput performance of GSS and PGSS can be measured by the overall shuffle throughput of the cluster and the average shuffle rate of each individual shuffle task, respectively. We use the metric of the overall shuffle throughput because it reflects the cluster overall bandwidth utilization along the time axis. The start time and finish time of the shuffle tasks are usually different from each other during the shuffle phase, but can also overlap. The average shuffle rate of each shuffle task, on the other hand, reflects how each individual shuffle task can benefit from the throughput increase of the the cluster.

The overall shuffle throughput is depicted as the cumulative completion ratio of the overall shuffle workload. Fig. 5 shows the results of RSS, GSS and PGSS in various benchmarks in the physical cluster. GSS and PGSS outperform RSS in all benchmarks, and PGSS even performs better than GSS in most cases. In the SequenceCount benchmark, GSS and PGSS have similar shuffle throughputs, which shows that the simple greedy method can also maximize the bandwidth utilization in some cases, as long as that it selects the fetch source node based on the estimation of the bandwidth allocation status. The overall shuffle throughput improvement is the result of maximizing the overall bandwidth utilization.

One thing to notice is that we can see the slightly longer tails in the shuffle CR line (e.g., RSS and GSS in Terasort and RSS in SequenceCount). It is because a few of the shuffle tasks would likely gain lower throughput at the beginning, and when some other tasks finish shuffling, they become stragglers. The largest throughput these remaining stragglers can attain depends on the bandwidth of the working links, but not the bandwidth of the whole cluster. As a result, the transfer rate slows down and we see the tails.



(a) Terasort

(b) InvertedIndex

(c) SequenceCount

(d) RankedInvertedIndex

Fig. 7. Reduce Completion Time of RSS, GSS and PGSS in Various Benchmarks with Different Numbers of Fetchers in Each Shuffle Task in the Physical Clusters

The comparison of average shuffle rates of the shuffle tasks between GSS, PGSS and RSS is made based on the speedup factor. The speedup is calculated by the average shuffle rate difference between GSS (or PGSS) and RSS over that of RSS. The results are shown in Fig. 6. GSS and PGSS gain relatively high shuffle rate speedups as compared to RSS in all benchmarks, which are as high as 7% and 12% in the Terasort benchmark. The speedups of PGSS are a bit higher than GSS, e.g., in Terasort and RankedInvertedIndex.

4.1.2 Completion Time

We also measure how much the job completion time can be decreased by maximizing the network bandwidth utilization with GSS and PGSS. The reduce completion time is the duration from the time when all the map tasks have finished to the time the job finishes. During the reduce completion time, no map task is running, and most of the tasks are doing the shuffle work at the first half of the time and doing the reduce work at the second half of the time. We did not observe the metrics of "shuffle completion time" because the tasks finish the shuffle work at different times and it is hard to define the endpoint of the shuffle completion time.

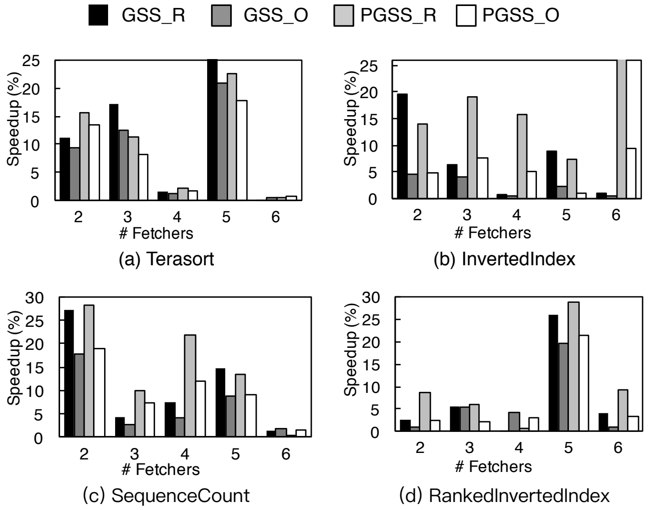
The reduce completion time of RSS, GSS and PGSS in various benchmarks with different numbers of fetchers in the shuffle tasks are shown in Fig. 7. Different numbers of fetchers will create different degrees of traffic congestion in the network. The case of only one fetch in each shuffle task is skipped because the sparse communication pattern will leave some of the links idle. The results show that PGSS takes significantly less time to complete the reduce than RSS. The exact speedup is depicted in Fig. 8 and will be described soon. The reduce completion time of RSS with different numbers of fetchers deviates from each other over a large value range without obeying following an obvious pattern, which indicates that the random source selection method cannot fully utilize the network bandwidth for different levels of network traffic congestions. It is not our work to obtain such an “optimal” fetcher number that generates the least reduce time because it seems unpredictable from the results of different benchmarks or just may not be unique (e.g., 4 fetchers and 6 fetchers in RankedInvertedIndex generate almost the same reduce time, and it is hard to tell the fetch number is an important impact factor for the same reduce time), which indicates that the obtaining the “optimal” fetcher number may be a false proposition. Besides, even if there is such an “optimal” fetcher number for RSS, it requires users multiple runs to obtain such number, and the best reduce completion time is narrowly shorter than the worst one with PGSS in the Terasort Case. While in the other benchmarks, PGSS always outperforms (or performs no worse than) RSS with any fetcher number. Nevertheless, the reduce completion time of GSS and PGSS (especially PGSS) is more flat with different numbers of fetchers in all benchmarks. It shows that BAShuffler can be applied to various traffic congestion statuses.

Fig. 8 depicts the reduce completion time speedup and the job overall completion time speedup of GSS and PGSS as compared to RSS with different numbers of fetchers in each shuffle task. The completion time speedup is calculated based on the time difference between GSS (or PGSS) and RSS over that of RSS. 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 by a remarkable margin. For example, in the RankedInvertedIndex benchmark with 5 fetchers, GSS and PGSS decrease the reduce completion time by 26% and 29%, respectively, and decrease the overall completion time by 19% and 21%, respectively. In some cases, the speedups of GSS and PGSS are not obvious (e.g., Terasort with 6 fetchers and RankedInvertedIndex with 4 fetchers) when the reduce completion time of RSS is already the minimum among all the fetcher settings.

4.2 Virtual Testbed

The virtual cluster is meant to evaluate the performance of BAShuffler in the environment when the network settings are unclear.

We deploy the virtual cluster on Amazon EC2, with 9 c3.xlarge instances and 10 c3.2xlarge instances running Amazon Linux Image [36]. The c3.xlarge instance is referred to as the *medium* instance and the c3.2xlarge instance the *large* instance below. The network topology and configuration of EC2 clusters are unclear, but the transfer rates of the access layer links are more or less stable. By our measure, the actual uplink and downlink bandwidth capacities of the medium instance are both about 720Mbps, and those of the large instance are about 1000 Mbps. One large instance is set up as both the name node of HDFS and the resource manager of YARN, and each of the remaining 18 instances, whether medium or large, is set up as both a data node of HDFS and a node manager of YARN.



(a) Terasort

(b) InvertedIndex

(c) SequenceCount

(d) RankedInvertedIndex

Fig. 8. Reduce Completion Time Speedup (Suffixed with _R) and Job Overall Completion time Speedup (Suffixed with _O) of GSS and PGSS as Compared to RSS in Various Benchmarks with Different Fetcher Numbers in the Physical Clusters

As the performance of BAShuffler in various benchmarks has been evaluated in the physical cluster and BAShuffler has similar performance in these benchmarks, in the virtual cluster, we use only one benchmark (i.e., SequenceCount, which has the largest input data size) for the evaluation for the reason of simplicity and avoiding redundancy.

4.2.1 Cluster Overall Shuffle Throughput

Similarly, the overall shuffle throughput of the cluster is described by the cumulative completion ratio of the overall shuffle workload. Fig. 9 depicts the results of the SequenceCount benchmark being executed in the virtual cluster, where BAShuffler delivers much higher shuffle throughput than RSS. A different observation from the physical cluster is witnessed: GSS is faster than PGSS in the virtual case. The reason is discussed as follows. The network bandwidth capacities of the links in the virtual cluster are much higher than those of in the physical one, with similar CPU rates. With the higher shuffle throughput, the length of the list of the pending fetch sources is smaller in the virtual cluster, as the fetch tasks finish faster. Although PGSS is designed to prevent the worse case that some heavy-loaded nodes are starved in the long period, which GSS may encounter, GSS is good enough to handle the scheduling of the pending sources of short length.

It leads us to the thought of the choice between GSS and PGSS. GSS performs well when the pending source list is of short length, which is either because the network throughput is high as compared to the data size that the CPU can generate, or because the shuffle size is relatively low (shuffle-trivial). With the setting that the network would be bottleneck for the job and the workload type is shuffle-heavy, by further considering the lower scheduling overhead in the large-scale cluster (which will be evaluated in Section 4.3.3, PGSS is preferred.

Although the underlying network settings are unclear, the experiment results show that BAShuffler can also im-

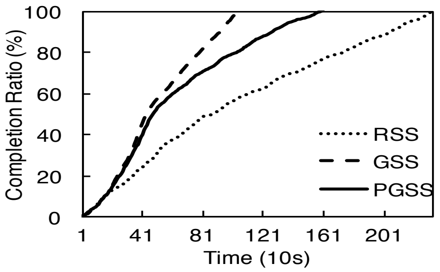


Fig. 9. Cumulative Completion Ratio of Shuffle Workload in the SequenceCount Benchmark along the Time in the Virtual Cluster

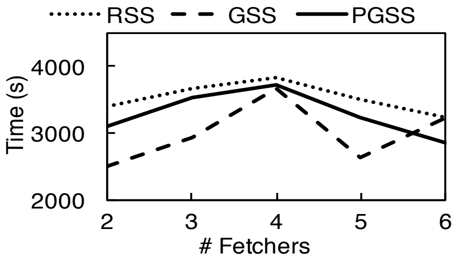


Fig. 10. Reduce Completion Time of the SequenceCount Benchmark with Different Fetcher Numbers in the Virtual Cluster

prove the bandwidth utilization in the EC2 virtual cluster by regarding the virtual network as a non-blocking switch.

4.2.2 Completion Time

The reduce completion of the SequenceCount benchmark with different fetcher numbers in the virtual cluster is shown in Fig. 10. Similar to the situation of the physical cluster, BAShuffler decreases the reduce completion time in different fetcher number settings. When the fetcher number is 2, GSS and PGSS decrease the reduce completion time by 26% and 8%, respectively. The curve of PGSS is more flat than the other scheduling algorithms and is always below that of RSS, which indicates that PGSS guarantees high shuffle performance regardless of the network congestion status. With 6 fetchers, GSS gives merely the same reduce completion time as RSS, but PGSS decreases the time by about 11% as compared to RSS.

4.2.3 Different Input Data Sizes

To evaluate the scalability of BAShuffler in processing large datasets, we run BAShuffler against different sizes of input data for the jobs. Fig. 11 shows the RSS and PGSS reduce completion time of the SequenceCount Job with different input data sizes in both the physical and the virtual testbeds. Note that the total numbers of nodes of the physical and virtual testbeds are different. PGSS performs better than RSS in both the testbeds regardless of the input data size. The completion time improvements range from 8% to 13% in the physical testbed and from 2% to 17% in the virtual one. The influence of the application-level shuffle scheduling is contributed more by the characteristics of the jobs and the varieties in the resource configurations (e.g., CPU, memory and network) of the cluster, rather than the size of the input data.

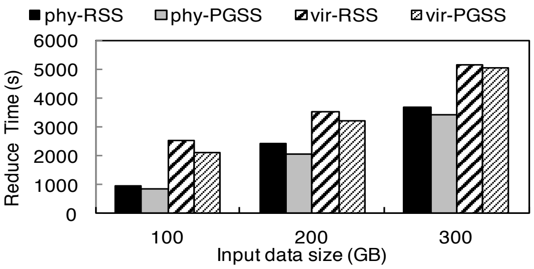


Fig. 11. Reduce Completion Time of SequenceCount with Different Input Data Sizes in the Physical and Virtual Testbeds

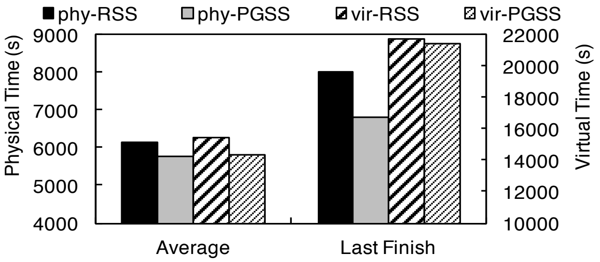


Fig. 12. Job Completion Time of Multiple Concurrent Jobs in the Physical and Virtual Testbeds

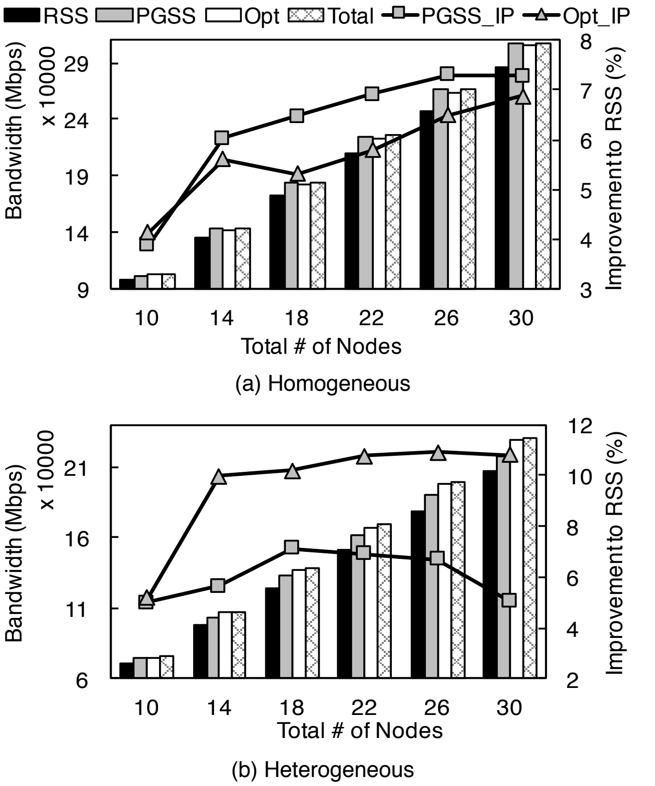
4.2.4 Multiple Concurrent Jobs

BAShuffler improves the concurrent job execution time by increasing the throughput of the concurrent shuffle tasks. We evaluate how BAShuffler can improve the job completion time when multiple jobs run concurrently in the shared cluster. We submit totally three SequenceCount jobs to the clusters at the same time. The container scheduler is the default Capacity Scheduler in YARN. Fig. 12 shows the average job completion time and the last job completion time of the concurrent jobs. Note that the times for the physical testbed and the virtual testbed are in different scales in the figure. PGSS decreases the average completion time and the last job completion time by about 6% and 15% in the physical testbed and by about 7% and 2% in the virtual testbed, respectively. In the virtual testbed, the second job completion time of PGSS is still 13% higher than that of RSS, but the final results at "2%". It may be because PGSS has the chance of pending the shuffle tasks of the third job as a penalty of achieving a maximum MMF allocation, while RSS is more fair for different jobs but gains poorer average completion time.

4.3 Simulation

It is hard to measure the performance of BAShuffler in real testbeds of different scales; we rely on simulations to provide some insight of algorithmic benefits and overheads, and the results are convincing. The different scheduling algorithms will generate scheduling decisions of the source nodes, and the bandwidth utilization of these scheduling decisions is calculated based on the MMF mechanism.

We classify the synthetic nodes into low-bandwidth nodes and high-bandwidth nodes. The bandwidth capacity of the uplink and the downlink of the low-bandwidth nodes is 512 Mbps and it is 1024 Mbps for the high-bandwidth



(a) Homogeneous

(b) Heterogeneous

Fig. 13. Bandwidth Utilization and Improvement (Suffixed with IP) of RSS, PGSS and the Online Optimal Algorithm (Opt) with Different Amount of Nodes in the Homogeneous Settings and the Heterogeneous Settings

nodes. Unless specified, otherwise, there is one shuffle task running on each node, and each shuffle task has 5 fetchers in total. Each shuffle needs to fetch data from all the other nodes in the cluster. One shuffle of the node will request to schedule all its fetches first before the next shuffle on another node requests to schedule its fetches.

All the tests about PGSS and the online optimal algorithm are executed 10 times, and 100 times for the tests about RSS due to its property of total randomness. The bandwidth utilization value is obtained as the mean value of all the runs. There is one shuffle task with 5 fetchers in each node. During the workload, at every time interval, the shuffle on a node will request to schedule the source nodes for its free fetchers before it is the turn for the shuffle on the next node in the sequence. When there is no free fetcher in a shuffle, the shuffle will assume that a random working fetcher has finished the fetch, release the corresponding fetch flow, and schedule a new pending source node for the new free fetcher. When the shuffle finishes fetching data from all the pending source nodes, a new shuffle will start on the same node to start to fetch data from all the other nodes again if it has free fetchers. Simulation results show that PGSS performs much better than RSS in bandwidth utilization with different numbers of nodes and different degrees of network heterogeneity.

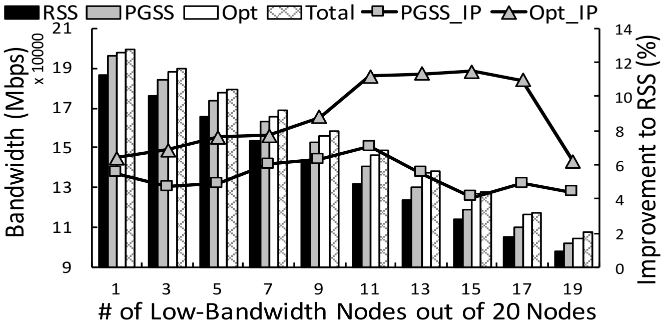


Fig. 14. Bandwidth Utilization and Improvement (Suffixed with IP) of RSS, PGSS and the Online Optimal Algorithm (Opt) with Different Degrees of Network Heterogeneity

4.3.1 Different Numbers of Nodes

We first evaluate the performance of PGSS and the online optimal algorithm in clusters of different scales, i.e., different numbers of nodes. Both the homogeneous and heterogeneous network modes are tested. In the homogeneous mode, all the nodes are high-bandwidth nodes, while in the heterogeneous mode, half of them are low-bandwidth nodes and the other half are high-bandwidth nodes. Fig. 13 shows the bandwidth utilization of the homogeneous and heterogeneous modes, respectively. The online optimal scheduling solution (Opt) is the maximum online result by traversing every possible permutation of the pending requests that have already seen by the scheduler. The total bar represents the bandwidth capacities of all the links in the cluster. Two solid curves stand for the improvement of PGSS and Opt as compared to RSS, respectively. The bandwidth utilizations of both PGSS and Opt are about 3.9% to 7.3% higher than RSS in the homogeneous mode and about 5.0% to 10.9% in the heterogeneous mode. An interesting result is that PGSS performs even slightly better than the online optimal algorithm in the homogeneous mode, which is possible as the online optimal algorithm only optimize the bandwidth utilization without considering the pattern of the future fetch flows, while PGSS labels the future fetch flows with the load count. In all cases, the bandwidth utilizations of PGSS are close to the total bandwidth capacities.

4.3.2 Different Degrees of Network Heterogeneity

We also evaluate the bandwidth utilization of different source scheduling algorithms with different degrees of network heterogeneity, i.e., different ratios of low-bandwidth nodes to high-bandwidth ones. We run the scheduling algorithms with totally 20 synthetic nodes, some of which are low-bandwidth nodes and the rest are high-bandwidth nodes. The result is shown in Fig. 14, where the left and the right ends of the horizontal axis stand for the higher degree in homogeneity and the middle part of the figure stands for the higher degree in heterogeneity. PGSS performs better than RSS in various heterogeneous networks, especially with higher degrees in heterogeneity, With higher degrees in heterogeneity, the PGSS bandwidth utilization increase can be as high as about 7%. This means that the application-level bandwidth-aware source selecting algorithm performs better in the heterogeneous networks.

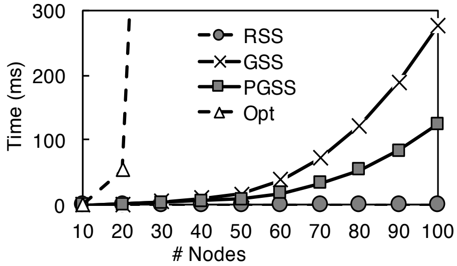


Fig. 15. Scheduling Overhead Time per Flow of the RSS, GSS, PGSS and Online Optimal (Opt) Algorithms with Different Number of Nodes in the Cluster

4.3.3 Scheduling Overhead

The scheduling overhead of BAShuffler is the time to figure out the source scheduling decision, is related to the number of total nodes and existing flows (whose number generally grows as the the number of nodes grows) in the cluster. Fig. 15 depicts the overhead time of RSS, GSS, PGSS as well as the online optimal scheduling algorithm with different numbers of nodes in the cluster. The scheduling overhead of RSS is taken for granted the lowest, which is not related to the number of nodes in the cluster. PGSS scales much better than GSS, as PGSS only considers the heaviest loaded nodes when estimating the network bandwidth utilization. Nevertheless, the scheduling overhead of BAShuffler is quite low comparing to the execution time of the jobs and the throughput improvement it can bring. While the overhead online optimal algorithm grows dramatically as the number of nodes increases, which is totally impractical in the real-life datacenters.

5 Related Work

Distributed Resource Management: Existing resource managing platforms for clusters, like YARN [5] and Mesos [8], allow distributed computing frameworks run on the cluster with fine-grained control of computation and storage resources. However, they do not have control over network resources in the cluster for tasks that produce a fair amount of traffic. A few scheduling techniques operate in the granularity of a task to improve resource scheduling by maintaining fairness across tasks [37], [38], or by achieving data locality (thus reducing the traffic workload) [1], [10], [39]. BAShuffler considers the network bandwidth utilization of the shuffle flows and schedules the source nodes of the flows to achieve a better utilization of the overall bandwidth.

Network Bandwidth Scheduling: Much work has been done on improving network performance and the fair share of network bandwidth in datacenter networks at the network level, either by individual flow scheduling [12], [18], [19], [28] or by scheduling the flows in terms of "coflow" [14], [15], [40], [41], [42], [43]. Specially, Chen et al. [43] analyzed the max-min fairness utilization of the coflow algorithm. In contrast to the network-leveled optimization, BAShuffler improves the shuffle performance at the application level by scheduling the sources of the TCP flows with GSS or PGSS and leave the sharing of bandwidth to the TCP natural behaviors. YARN with BAShuffler can easily be ported to any commercial cluster without changing the underlying network.

A work closely related to this paper is Orchestra [13], which reduces the shuffle completion time by assigning a weight to a set of flows depending on the data size at the application level. More TCP flows will be created for the shuffle task that has a larger volume of data to transfer. It assumes that each TCP flow will get approximately the same bandwidth in the congested network link due to the TCP fairness policy, and thus the shuffle task will obtain a bandwidth proportional to its weight in this link. However, it is often not true that each TCP flow will have an equal share of the bandwidth in the network, especially in heterogeneous network environments. The throughput of a shuffle task cannot simply be speculated based on the number of concurrent TCP flows in this case. BAShuffler applies the max-min fairness behavior of TCP communications to estimate the bandwidth share of the TCP flows, which is a better approximation of the overall bandwidth allocation than merely using the number of TCP flows. Sylvain et al. [44] proposes and compares several data transfer scheduling policies for the shuffle operation under bandwidth constrains, which has the potential to be applied in the MapReduce-specific network.

Online Scheduling: Compared to other online scheduling algorithms [32], [33], [45], [46], BAShuffler can predict the approximate arrival pattern of the pending sources based on the semantic of the shuffle, and thus can apply the partially greedy source selection method to maintain the performance in the long term.

6 Conclusion

In this paper, we demonstrate the underutilization problem of network bandwidth by the shuffle phase in both homogeneous and the heterogeneous network settings, by analyzing the max-min fairness behavior of the underlying TCP network. We present BAShuffler in YARN to improve the shuffle performance by selecting the source nodes of the shuffle flows that can maximize the overall bandwidth utilization in the cluster. GSS and PGSS are proposed to maintain the high bandwidth utilization. BAShuffler significantly increases the shuffle performance in both physical and virtual clusters, especially where the network is heterogeneous.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] A. Thusoo et al., "Hive: a warehousing solution over a map-reduce framework," *Proceedings of VLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[3] Y. Yu et al., "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 1–14.

[4] M. Armbrust et al., "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[5] V. K. Vavilapalli et al., "Apache hadoop yarn: Yet another resource negotiator," in *SOCC*. ACM, 2013, p. 5.

[6] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 2–2.

[7] T. White, *Hadoop: The Definitive Guide, 4th Edition*. "O'Reilly Media, Inc.", 2015.

[8] B. Hindman et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[9] S. Kavulya et al., "An analysis of traces from a production mapreduce cluster," in *The 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, May 2010, pp. 94–103.

[10] M. Zaharia et al., "Improving mapreduce performance in heterogeneous environments," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, no. 4, 2008, p. 7.

[11] K. Kant, "Data center evolution: A tutorial on state of the art, issues, and challenges," *Computer Networks*, vol. 53, no. 17, pp. 2939–2965, 2009.

[12] A. Shieh et al., "Sharing the data center network," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011, pp. 309–322.

[13] M. Chowdhury et al., "Managing data transfers in computer clusters with orchestra," *Proceedings of the ACM Conference on SIGCOMM*, vol. 41, no. 4, pp. 98–109, 2011.

[14] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the ACM Conference on SIGCOMM*, New York, NY, USA, 2014, pp. 443–454.

[15] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proceedings of the ACM Conference on SIGCOMM*, New York, NY, USA, 2015, pp. 393–406.

[16] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA, 2014, pp. 127–144.

[17] F. Liang and F. C. M. Lau, "Smapreduce: Optimising resource allocation by managing working slots at runtime," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2015, pp. 281–290.

[18] A. Greenberg et al., "Vl2: a scalable and flexible data center network," in *Proceedings of the ACM Conference on SIGCOMM*, vol. 39, 2009, pp. 51–62.

[19] L. Popa et al., "Faircloud: sharing the network in cloud computing," in *Proceedings of the ACM Conference on SIGCOMM*, 2012, pp. 187–198.

[20] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

[21] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 314–329.

[22] D. P. Bertsekas and R. G. Gallager, *Data networks*. Prentice-Hall International, 1992, vol. 2.

[23] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN systems*, vol. 17, no. 1, pp. 1–14, 1989.

[24] F. P. Kelly, A. K. Maulloo, and D. K. Tan, "Rate control for communication networks: shadow prices, proportional fairness and stability," *Journal of the Operational Research society*, pp. 237–252, 1998.

[25] M. Vojnović, J.-Y. Le Boudec, and C. Boutremans, "Global fairness of additive-increase and multiplicative-decrease with heterogeneous round-trip times," in *INFOCOM*, vol. 3, 2000, pp. 1303–1312.

[26] M. Alizadeh et al., "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the ACM Conference on SIGCOMM*, New York, NY, USA, 2014, pp. 503–514.

[27] R. Niranjan Mysore et al., "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *Proceedings of the ACM Conference on SIGCOMM*, New York, NY, USA, 2009, pp. 39–50.

[28] H. Ballani et al., "Towards predictable datacenter networks," in *Proceedings of the ACM Conference on SIGCOMM*, vol. 41. ACM, 2011, pp. 242–253.

[29] M. Alizadeh et al., "pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the ACM Conference on SIGCOMM*, New York, NY, USA, 2013, pp. 435–446.

[30] N. Kang et al., "Optimizing the "one big switch" abstraction in software-defined networks," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, New York, NY, USA, 2013, pp. 13–24.

[31] A. Phanishayee et al., "Measurement and analysis of tcp throughput collapse in cluster-based storage systems," in *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 175–188.

[32] X. Wu, R. Srikant, and J. R. Perkins, "Scheduling efficiency of distributed greedy scheduling algorithms in wireless networks," *Transactions on Mobile Computing*, vol. 6, no. 6, pp. 595–605, 2007.

[33] J. Sgall, "On-line scheduling," in *Online algorithms*. Springer, 1998, pp. 196–231.

[34] Hku gideon-ii cluster. [Online]. Available: http://i.cs.hku.hk/%7Eclwang/Gideon-II/

[35] F. Ahmad et al., "Puma: Purdue mapreduce benchmarks suite," 2012.

[36] "Amazon ec2 instance types." [Online]. Available: http://aws.amazon.com/ec2/instance-types/

[37] M. Zaharia et al., "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. New York, NY, USA: ACM, 2010, pp. 265–278.

[38] M. Isard et al., "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2009, pp. 261–276.

[39] F. Ahmad et al., "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 1–13. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ahmad

[40] F. R. Dogar et al., "Decentralized task-aware scheduling for data center networks," in *Proceedings of the ACM Conference on SIGCOMM*, New York, NY, USA, 2014, pp. 431–442.

[41] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, New York, NY, USA, 2015, pp. 294–303.

[42] Y. Zhao et al., "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *IEEE Conference on Computer Communications (INFOCOM)*, April 2015, pp. 424–432.

[43] L. Chen et al., "Optimizing coflow completion times with utility max-min fairness," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

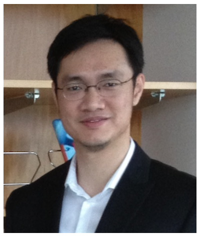
[44] S. Gault and C. Perez, "Dynamic scheduling of mapreduce shuffle under bandwidth constraints," in *European Conference on Parallel Processing*. Springer, 2014, pp. 117–128.

[45] B. Awerbuch, S. Kutten, and D. Peleg, "Competitive distributed job scheduling (extended abstract)," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing (STOC)*, New York, NY, USA, 1992, pp. 571–580.

[46] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. cambridge university press, 2005.

Feng Liang is currently a PhD candidate in Department of Computer Science, The University of Hong Kong. He received his bachelor's degree in software engineering in Nanjing University in 2012. His research interests are mainly on distributed file systems, distributed computing, and machine learning. He is recently undertaking the research project on distributed deep learning. [homepage] i.cs.hku.hk/%7Efliang

Francis C.M. Lau received his PhD in computer science from the University of Waterloo in 1986. He has been a faculty member of the Department of Computer Science, The University of Hong Kong since 1987, where he served as the department chair from 2000 to 2005. He is now Associate Dean of Faculty of Engineering, the University of Hong Kong. He was a honorary chair professor in the Institute of Theoretical Computer Science of Tsinghua University from 2007 to 2010. His research interests include computer systems and networking, algorithms, HCI, and application of IT to arts. He is the editor-in-chief of the Journal of Interconnection Networks. [homepage] i.cs.hku.hk/%7Efcmlau



Heming Cui is an assistant professor in Computer Science of HKU. His research interests are in operating systems, programming languages, distributed systems, and cloud computing, with a particular focus on building software infrastructures and tools to improve reliability and security of real-world software. [homepage] i.cs.hku.hk/%7Eheming

Cho-Li Wang is currently a Professor in the Department of Computer Science at The University of Hong Kong. He graduated with a B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1985 and a Ph.D. degree in Computer Engineering from University of Southern California in 1995. Prof. Wangs research is broadly in the areas of parallel architecture, software systems for Cluster computing, and virtualization techniques for Cloud computing. His recent research projects involve the development of parallel software systems for multicore/GPU computing and multi-kernel operating systems for future manycore processor. Prof. Wang has published more than 150 papers in various peer reviewed journals and conference proceedings. He is/was on the editorial boards of several scholarly journals, including IEEE Transactions on Cloud Computing (2013-), IEEE Transactions on Computers (2006-2010). [homepage] i.cs.hku.hk/%7Eclwang