Improving Shuffle Network Bandwidth Utilization by Application-Level Flow Scheduling

Feng Liang,Francis C. M. Lau,Heming Cui,and Cho-Li Wang

Abstract—Shuffle operations invoke many network flows across computer nodes and consume significant network bandwidth. Network bandwidth is usually largely under-utilized for distributed applications, especially the shuffle-heavy ones. Worse, in practice, when network bandwidth capacities of computer nodes vary, the under-utilization problem is more pronounced. We first present BAShuffler, a bandwidth-aware shuffle scheduler that can maximize the overall network bandwidth utilization in shuffle operations. We then demonstrate how BAShuffler fully utilizes the network bandwidth in various network settings and network traffic scenarios. Compared to existing network-level approaches, BAShuffler takes the first step to adopt an application-level approach that has the full picture of shuffle operations and schedules flow source nodes with a precise bandwidth allocation estimation. We evaluate BAShuffler with a variety of realistic benchmarks in testbeds comprising a physical, heterogeneous cluster and an EC2 virtual cluster, respectively. Experiment results show that BAShuffler significantly improves Hadoop shuffle throughput and decreases shuffle completion time by about 30%.

Index Terms—Distributed Platform, MapReduce, Shuffle, Network Scheduling.

1 Introduction

The shuffle operation is pervasive in MapReduce-like distributed platforms [1], [9], such as the "*join*" operation in distributed database systems [2], [3], [4], the "*reduce*" task in MapReduce systems [1], [5] and the "*aggregateByKey*" operation in Spark [7]. The shuffle operation requires large amounts of network bandwidth and is observed to dominate the job completion time in datacenters. A study on Facebook MapReduce workloads [6] suggests that about 10% of the large jobs are shuffle-heavy, whose shuffle operation accounts for 46% of the total data movement and 53% of the total job completion time. High network bandwidth utilization is critical for reducing the completion time of shuffle-heavy jobs.

Unfortunately, existing distributed platforms (e.g., YARN [5]) mainly concentrate on managing node-monopolized resources (e.g., CPU cores and memories), and the network bandwidth utilization is often ignored. For many shuffle-heavy tasks, the communication is intensive. Their performance is heavily affected by the throughput of network flows, which is mainly determined by the available network bandwidth of the connection on both endpoints of the source and the destination. However, managing the network bandwidth is difficult because it is the shared resource for a group of computer 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.

To improve the network bandwidth utilization, existing distributed platforms [5], [7] adopt a random flow source node selection policy. The random policy do not consider the diverse network bandwidth usage of flows, but only try to evenly distribute the workload of the network by randomly selecting a source to fetch the map output data. If links connecting all the nodes in the cluster are more or less equal in terms of bandwidth capacity and if the number of network connections is large, this random source selection policy would prevent some nodes becoming a bottleneck. However, without monitoring the actual bandwidth usage of network flows, this random source scheduling approach often under-utilizes the network bandwidth when some nodes are congested by unpredictable flow jams [13].

Worse, when the network bandwidth capacities of the links are heterogeneous, the bandwidth under-utilization problem is more pronounced. We discuss this finding by analyzing with examples in Section 3.5. A heterogeneous network environment is common in practice for distributed platform users. For example, Hadoop is designed as a general distributed processing platform for clusters comprising off-the-shelf machines [1]. Physical Hadoop clusters could be fit with a variety of servers and network devices of different configurations [10]. Even in enterprise datacenters, heterogeneity exists during the long period of machine upgrading [11].

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. A central scheduler at the application level sees the full picture of the shuffle operation and has the potential to achieve the optimal scheduling results based on precise estimations of the network bandwidth allocation. Although some work concentrates on the network level [12], [13], [14], [15], the network level models cannot fully capture the behavior of the shuffle operation, e.g., the dynamic and casual arriving time of the fetch flows when a map task finishes. We will discuss in detail the rationality of improving the shuffle performance by scheduling the sources of the

• *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

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 makes use of the max-min fairness behavior of TCP communication and selects the appropriate source nodes of the TCP fetch flow so that the network bandwidth utilization is maximized. We implement BAShuffler based on YARN and apply it in a variety of benchmarks.

We thoroughly evaluate BAShuffler and verify that BAShuffler significantly increases the shuffle performance. Experiment results on a physical cluster and an EC2 virtual cluster show that BAShuffler can increase the shuffle throughput and reduce the total job completion time by as much as about 30% as compared to the original YARN, especially for shuffle-heavy jobs. We also use simulations to mimic clusters of various scales. The simulation results show that comparing to the random-source-selection method adopted by YARN, BAShuffler greatly improves the heterogeneous network bandwidth utilization, with low scheduling overheads.

We have done some initiatory work that shows how the application-level source selection policy can improve the bandwidth utilization in shuffle operations [16]. In this paper, we present the complete work with the following major contributions:

- We explored in depth the rationality and benefits of improving the shuffle bandwidth utilization by scheduling flow source nodes at the application level.
- We presented efficient source selection algorithms, GSS and PGSS, which maximize the network bandwidth utilization in shuffle operations. GSS and PGSS work in various network settings and communication traffic scenarios.
- We implemented the system BAShuffler, which schedules shuffle sources at the application level, and verified that it greatly improves the performance of shuffle-heavy jobs in diverse cluster environments. BAShuffler is embedded seamlessly in distributed platforms, and existing MapReduce-like applications run on it without any modification in greatly improved bandwidth utilization.

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. We present the design and implementation details of BAShuffler and show how to apply BAShuffler to maximize the network bandwidth in different scenarios 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 Motivation and Discussion

In this section, we present an overview of 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 Shuffle in YARN

YARN, the latest Hadoop [8], manages the computing resources in the cluster as a collection of containers, each of which consists of memories and CPU cores. Jobs are sub-divided into tasks according to a certain computing paradigm, which are then assigned to their respective allocated containers owning a required amount of memories and CPU cores.

Although much work [17], [18] 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 outputs of the map interface as inputs 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 interface executes, needs to fetch the map output data corresponding to a specific key range from different nodes for later reducing, which is called a "**shuffle**". Each fetch is seen as a TCP flow, which sends a HTTP request to the map node to transfer map outputs of a specific key partition to the reduce node. There are usually many fetches from different source nodes in a shuffle. A shuffle is a many-to-one TCP communication instance, and all shuffles together of a MapReduce job make up a many-to-many one. 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 source nodes to transfer 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 when many shuffles are executing, and reduce the probability of the traffic congesting in the uplink of some popular source nodes.

RSS is widely used in the popular distributed platforms as the major shuffle source selection algorithm [5], [7]. However, due to the dynamic task running environment, RSS can suffer from the case that network flows concentrate on a specific set of links at some period. Moreover, as we have commented, RSS cannot fully utilize the network bandwidth when the network bandwidth capacities of network links connecting the nodes are different, i.e., in a heterogeneous network environment. In such an environment, even with the same number of flows on the links, links of low bandwidth capacity can suffer from network congestion, becoming the performance bottleneck, while links of high bandwidth capacity keep quite a portion of bandwidth idle.

A shuffle flow scheduler that can select proper source nodes to fetch data is desired 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.

Scheduling individual network flows is not enough for the shuffle. At the network level, ideas such as performance isolation [19] and fair sharing of network resources [12], [20] can offer performance improvements for each single fetch flows. However, as the fetch 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 [14].

Network-level models can only see a part of of the shuffle operation. Network-level models such as the "coflow" model [14], [15] have 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 keep pending until there are available fetch workers later. Network-level models (or coflow here) are only aware of the existence of the flows created, but not the remaining flows of the same shuffle pending at the application level. With this limitation, the shuffle operation cannot exactly fit into the coflow model. Efforts on optimizing such network-level models are not directly related to improving the performance of shuffle operations.

A application-network cross-layer solution can be too complex and complicated. 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 needs to gather (or distribute) a large amount of information from (or to) both the application level and the network level. The overhead of communication between the application level and the network level could be prohibitive. Such a cross-layer approach would violate the principle of the isolation of the application level and the network level [21].

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). Although it does not try to improve the underlying network, it can observe and predict the behavior and performance of the network, and then make the shuffle scheduling decisions based on these network 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 [22]. The MMF of TCP has been extensively analyzed and verified [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.

In a MMF 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. Of all the data flows sharing the bottleneck link, this 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 flows are known in advance. The output is the allocated bandwidth of every data flow. 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. Links with the least average bandwidth share are saturated links, and this least bandwidth share is the bandwidth allocated to these flows on the saturated links. The second step is to remove the saturated links and the flows that have been just allocated the bandwidth, and update the available bandwidth capacities of the remaining links. For each remaining links that is on the path of a removed flows, the bandwidth occupied by the removed flow is subtracted from the link's available bandwidth capacity. Finally, the first and second steps are repeated until every flow has been allocated a bandwidth share.

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 [19], [26], [27], it is practical to simplify the datacenter fabric as a non-blocking switch [13], [14], [15], [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. Though we assume the non-blocking switch network, it is not necessary that MMF and BAShuffler should be limited to such a network. MMF is a general estimation on TCP networks, and as long as BAShuffler obtains the MMF allocation, BAShuffler is applicable to general TCP networks.

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.

BAShuffler improves 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 is implemented seamlessly in YARN and existing MapReduce jobs can run on YARN without changing any code in the presence of BAShuffler.

3.1 Design of BAShuffler

3.1.1 Design Considerations

To improve 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 will be improved with faster shuffles.

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 memories or CPU cores, for the following two reasons: a) the network bandwidth occupied by a 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, and a network-bandwidth-encapsulated container can prohibit other flows from using the bandwidth during the lifetime of the container, even when the task is not working on the shuffle. Therefore, BAShuffler will not change the allocation mechanism of resource containers for the tasks. Instead, BAShuffler focuses on the shuffle phase of reduce tasks, 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 fetch flows and the map task straggler that fails to have all map outputs ready. These factors are related to the dynamic environment of distributed platforms. Some of them can be avoided by a better resource isolation mechanism. For instance, a good resource isolation mechanism should make sure that the CPU time slices guaranteed for sending the flow data will be stolen by other compute-intensive jobs. BAShuffler will 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 characteristics, but generally need not be large (as suggested by the incast problem). According to some of the Hadoop best practices (e.g. [8]), tuning the number of fetchers is not a major performance consideration. Considering that there can be multiple task containers in one node, we assume 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 and input sizes, we do not bother with the impact of the fetcher number and simply used Hadoop's default number, which is 5.

Another important factor is the communication pattern of fetch flows. The communication pattern refers to the mappings from flow sources to flow destinations. Different communication patterns can lead to different bandwidth utilization status, which we will illustrate in later sections. BAShuffler works 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

BAShuffler is embedded in YARN. The structure of BAShuffler components and the shuffle-related information maintained by them are shown in Fig. 1. The arrow lines represent remote procedure calls (RPC) between components in different nodes. The resource manager, the node manager and the application master are the conventional terms in YARN, where the resource manager controls the resource allocation in the cluster, the node manager controls resources in each node, and the application master coordinates the task assignment and execution of a distributed job (T1). BAShuffler is a centralized scheduler in the resource

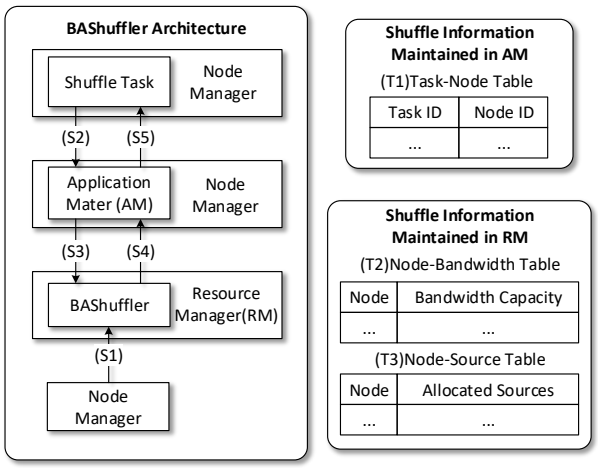


Fig. 1. The Architecture of BAShuffler and Shuffle-Related Information Maintained in Components

manager and makes source scheduling decisions based on bandwidth capacities of links (T2) and the scheduling status of the whole cluster, e.g., the records of the source and destination nodes of the scheduled flows (T3), with the fact that this information is easily at hand of the resource manager.

At startup, node managers register with BAShuffler the bandwidth capacity of the uplinks and downlinks of the nodes (S1). Users can obtain the capacities of these access layer links of the nodes either from the specification of the cluster configuration or simply by a bandwidth evaluation tool.

When a shuffle task needs to schedule a new fetch, with the node executing the shuffle task as the destination node, it initiates a request to the application master for a scheduled fetch (S2). The request contains the shuffle information, including the shuffle task ID, the list of pending source nodes for fetching data and the amount of source nodes to request. 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 a heartbeat from the application master to the resource manager, besides the original container allocation requests of YARN, the application master also sends those shuffle source scheduling requests to BAShuffler (S3). BAShuffler applies a shuffle scheduling policy (PGSS or GSS) to select sources of shuffle tasks, which will be introduced soon. When the source nodes are selected, BAShuffler will update the current scheduling status and the result is transferred back to the shuffle task in the reversed path (S4 and S5).

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

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

Algorithm 1: Greedy Source Selection and Partially Greedy Source Selection

Input : $Sources$: source nodes to select;
$Pattern$: sources and destinations of the allocated flows in the cluster;
Output: $Selected$;

```
if GSS then
    Nodes ← Sources
end
else if PGSS then
    Nodes ← the heaviest-loaded source nodes in Sources;
end
MaxBandwidth ← 0;
foreach Source ∈ Nodes 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
if PGSS then
    update the load count of Selected;
end
add Selected to Pattern;
```

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 that after selecting it, the MMF bandwidth of the whole cluster is no less than that when any other pending source node is selected. We call this shuffle scheduling policy Greedy Source Selection, or GSS. Algorithm 1 presents the procedure of GSS.

GSS selects the source node that currently maximizes the cluster MMF bandwidth. GSS is an effective method when it is not the worst case: the selections of shuffles concentrate 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 other online scheduling algorithms [32], [33], GSS does not predict the arrival pattern of coming source nodes. It makes sure that the maximum transfer throughput of the whole cluster is achieved at the moment it makes the

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

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

scheduling decision, but it cannot guarantee the maximum average throughput of the cluster over a long period of time.

A better shuffle source scheduling algorithm is possible if it can predict the arrival pattern of the source nodes in the future. There is, however, almost no way to know the arrival pattern of the shuffle tasks in advance as when and where a shuffle task starts depends on the allocation of the reduce container, which is a stochastic process. Nevertheless, after a shuffle has started, the future pairs of the sources and the destination of fetch flows can be approximately predicted. The destination of these fetch flows is the node where the shuffle task executes, and the sources are the nodes where the map tasks of the same job execute (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 adjust GSS with future prediction to what we call Partially Greedy Source Selection (PGSS), whose algorithm is also shown in Algorithm 1. PGSS marks each node with a *load count*, which indicates how many fetch flows will be created by this source nodes in the 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 future pending source node by the number of map tasks of in that source node. The source nodes with the highest load count is labeled as the heaviest-loaded nodes. When PGSS needs to select a source from the pending nodes, it zeros in on the set of heaviest-loaded nodes first ("partial") and selects the one that gives that maximum MMF bandwidth utilization ("greedy").

3.4 Comparing GSS and PGSS

PGSS may not deliver larger overall bandwidth utilization as compared to GSS, as PGSS only selects source nodes from the heaviest-loaded sources, which are in a subset of the domain of GSS. When the pending source list is long, GSS may concentrates its selection on a set of sources that are not heavy-loaded first, in order for the maximum MMF allocation. But later in the long run, the remaining sources are heavy-loaded and the throughput of GSS slows down and the shuffle progress starts to drag. PGSS better resolves the dragging case.

4.2

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 set of pending sources of size M at the same time. 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 MMF 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 that of the online optimal algorithm. Generally, PGSS suffers even less scheduling overhead than GSS. It is verified in Section 4 that the PGSS gains performance close to the online optimal but with a much smaller scheduling overhead.

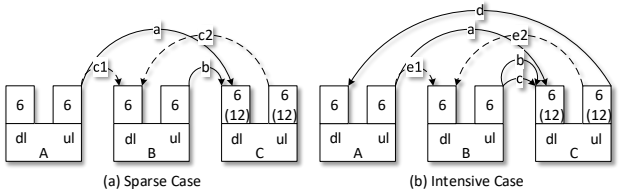


Fig. 2. Scenarios of Selecting the Source in Uneven Flow Pattern

The choice between GSS and PGSS will be discussed in Section 4.2 accompanied with experiment results. In short, PGSS is the better choice for heavily communication-intensive scenarios (traffic predictable) and large-scale clusters (lower overhead). While GSS is more suitable for higher bandwidth networks (so that pending source list is short) and smaller-scale clusters.

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 job 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.

3.5 Applying GSS/PGSS

We illustrate how GSS or PGSS are applied when selecting a source based on the MMF allocation in both homogeneous and heterogeneous network settings, which are distinguished by whether the capacities of the access layer links are the same or not, 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. We assume that the bottleneck links of the TCP flows are the access links to computer nodes. Since GSS

TABLE 2
MMF Flow Allocation of the Sparse Case 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 3
MMF Flow Allocation of the Intensive Case in 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

and PGSS are similar except that PGSS selects source nodes from the heaviest-loaded nodes, we illustrate the benefit of MMF of estimation in BAShuffler by assuming all nodes are already heaviest-load nodes. GSS behaves exactly the same as PGSS then, and for simplicity, we only discuss about applying PGSS.

Fig. 2 depicts two scenarios of the uneven flow pattern, where uneven means that the numbers of flows into or out of the nodes are different, and even otherwise. In the sparse case, some 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 all 6; whereas in the heterogeneous setting, their 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 data. Assume that both Node A and Node C are the heaviest-loaded nodes.

3.5.1 Homogeneous Network

Different selection decisions lead to different flow bandwidth allocations and overall bandwidth utilizations. 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 the status before the source selection. In the sparse case (Fig. 2), PGSS selects Node C as the source, which offers 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.

Applying MMF to estimate the bandwidth allocation can help make better decisions when selecting the source nodes at the application level. In the intensive case (Fig. 2), if we assume that the upload (download) rates of 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.

TABLE 4
MMF Flow Allocation of Uneven 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

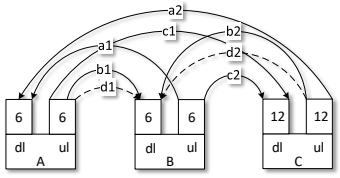


Fig. 3. Selecting the Source in the Even Flow Pattern

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

3.5.2 Heterogeneous Network

In the heterogeneous network setting, the randomly source selection policy can even have an 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. 2 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), as 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 is critical for high bandwidth utilization.

Fig. 3 depicts the scenario 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, 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.

Overall, neither RSS nor selecting the source based on the number of flows on the link makes the right decision that maximizes the bandwidth utilization in the MMF network. However, by applying the knowledge of the bandwidth capacities of the access layer links and the TCP flow communication patterns (sources and destinations of the flows), PGSS

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

can easily find out the proper source node to achieve the maximum transfer throughput of the flows by the behavior of MMF.

3.6 Implementation Details

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

Pre-Scheduling: Usually, the shuffle task has only one free fetcher at a time, and the shuffle requests for a scheduled source from BAShuffler every time (Fig. 1). For every source request from the shuffle task to 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 free fetchers ask for the sources. BAShuffler uses the *pre-scheduling* strategy, which requests a number of the scheduled sources from BAShuffler at a time, and when a fetcher needs a source, the shuffle task can return one immediately. Although the actual flow pattern may lag slightly behind the scheduling pattern stored in BAShuffler when pre-scheduling, the pattern will quickly catch up when the scheduled sources in the cache of the shuffle task 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 users. Furthermore, we also evaluate the scalability of BAShuffler in clusters of different scales and degrees of heterogeneity by simulation.

4.1 Physical Testbed

We run BAShuffler in a physical testbed, the Gideon-II cluster [34] in the heterogeneous network configuration. The physical testbed is a cluster containing 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

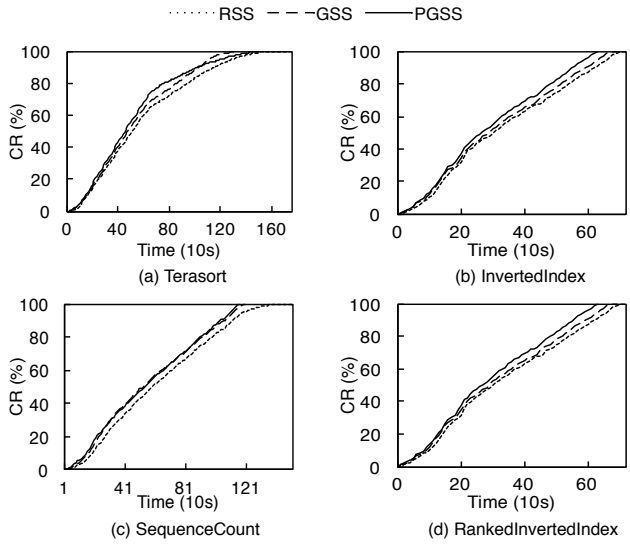


Fig. 4. Cumulative Completion Ratio (CR) of the Overall Shuffle Workload of RSS, GSS and PGSS in Various Benchmarks along the Time in the Physical Testbed

data nodes of HDFS and the node managers of YARN. Each node is equipped with 2 quad-core, 32 GB DDR3 memory and 2 $\times$ 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 capacity of the uplinks and downlinks of 8 nodes is manually limited to 160 Mbps, by using the traffic control tool "*tc*", and the remaining 8 nodes keep to their physical uplinks and downlinks 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 BAShuffler when the shuffle workload can saturate the network most of the time. BAShuffler performs almost the same in shuffle-light applications in the benchmark suite and we leave them out in the experiment results. The sizes of the datasets of the benchmarks are listed in Table 6.

BAShuffler is configured in advance to know 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), the number of reduce tasks of each job is configured to be the total number of node managers in the cluster, and other Hadoop configurations keep their default values. 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 sub-sections 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. The results will be introduced later together with the those of the virtual cluster.

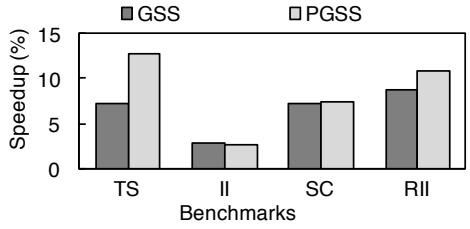


Fig. 5. GSS and PGSS Speedup of Average Shuffle Rate of Each Shuffle Task in comparison to RSS in Benchmarks Terasort (TS), InvertedIndex(II), SequenceCount (SC) and RankedInvertedIndex (RII) in the Physical Testbed

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 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 cluster.

The overall shuffle throughput is depicted as the cumulative completion ratio of the overall shuffle workload. Fig. 4 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 improved overall bandwidth utilization.

One thing to notice is that we can see 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 unfortunately gain lower throughput at the beginning. When some other tasks finish shuffling, they still cannot increase the throughput and become stragglers. The reason is that 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 is slow and we see the tails.

The comparison of the average shuffle rates of 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. 5. GSS and PGSS gain relatively high shuffle rate speedups as compared to RSS in all benchmarks, which are about 7% and 12% in the Terasort benchmark. The speedups of PGSS are a bit higher than GSS, e.g., in Terasort (by 5%) and RankedInvertedIndex (by 3%).

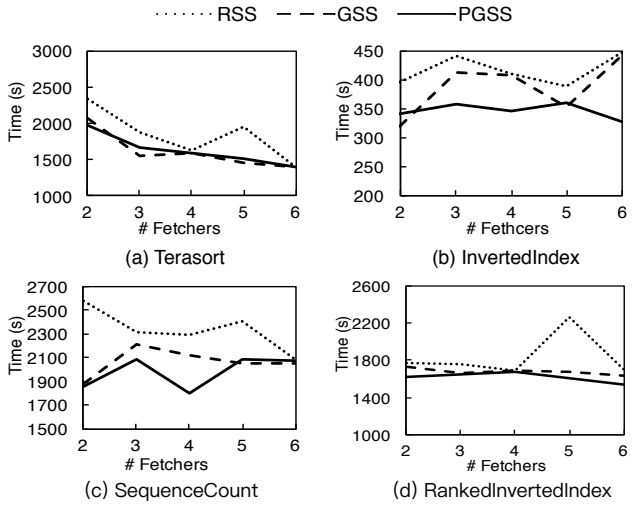


Fig. 6. Reduce Completion Time of RSS, GSS and PGSS in Various Benchmarks with Different Numbers of Fetchers in Each Shuffle Task in the Physical Testbed

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 time period between the time when all the map tasks have finished and 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.

Reduce Completion Time: The reduce completion time of RSS, GSS and PGSS in various benchmarks with different numbers of fetchers in shuffle tasks are shown in Fig. 6. 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. 7 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 RSS is sensitive to different levels of network traffic congestions. The reduce completion time of GSS and PGSS is more flat with different numbers of fetchers in all benchmarks. Except in Terasort, where PGSS and GSS are close in reduce completion time, PGSS generally performs better than GSS with different fetcher numbers in the other benchmarks. PGSS is more applicable to various traffic congestion statuses.

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

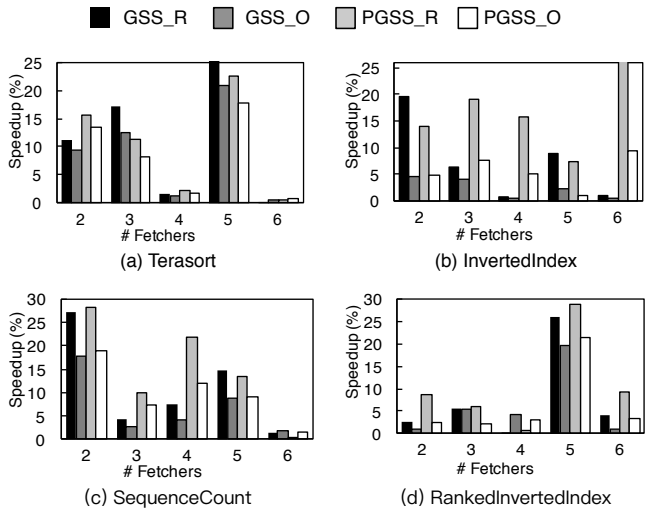


Fig. 7. Reduce Completion Time Speedup (Suffixed with _R) and Job Overall Completion time Speedup (Suffixed with _O) of GSS and PGSS as Compared to RSS in Various Benchmarks with Different Fetcher Numbers in the Physical Clusters

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's worst result is no worse than the best one of RSS with any fetcher number.

Overall Completion Time: Fig. 7 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 the time of RSS. As the benchmarks are reduce-heavy, where the shuffle phase occupies 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. We also note that 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 testbed is meant to evaluate the performance of BAShuffler in the environment when the network settings are unclear.

We deploy the virtual testbed on Amazon EC2, with a cluster consisting of 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.

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. 8 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 for a 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 think 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 improve 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 time of the SequenceCount benchmark with different fetcher numbers in the virtual cluster is shown in Fig. 9. 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

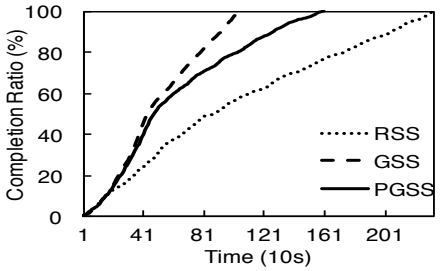


Fig. 8. Cumulative Completion Ratio of Shuffle Workload in the SequenceCount Benchmark along the Time in the Virtual Cluster

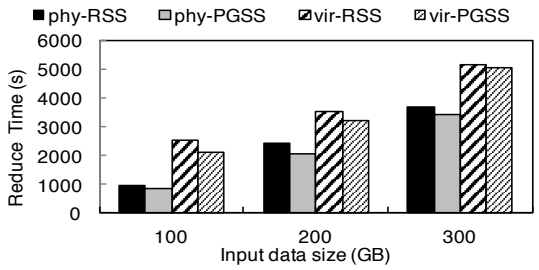


Fig. 10. Reduce Completion Time of SequenceCount with Different Input Data Sizes in the Physical and Virtual Testbeds

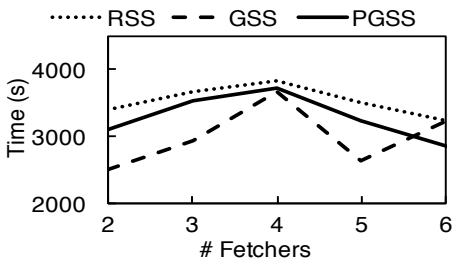


Fig. 9. Reduce Completion Time of the SequenceCount Benchmark with Different Fetcher Numbers in the Virtual Testbed

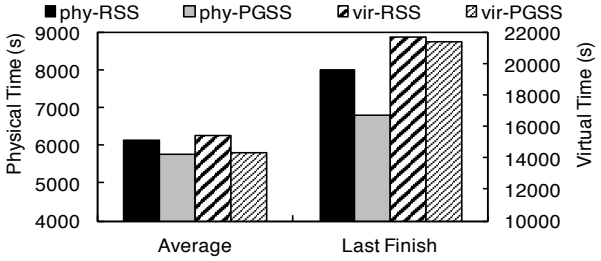


Fig. 11. Job Completion Time of Multiple Concurrent Jobs in the Physical and Virtual Testbeds

that of RSS, which is the same as in the physical cluster. As discussed in the previous sub-section, GSS generally performs better than PGSS because of the faster network transfer rate over the CPU rate. In the case with 6 fetchers, when the network is more congested, 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

The influence of the application-level shuffle scheduling is contributed by the characteristics of the jobs, the varieties in the resource configurations (e.g., CPU, memory and network) of the cluster, as well as the size of the input data. To evaluate the scalability of BAShuffler in processing large datasets, we run BAShuffler against different sizes of input data for the jobs. Fig. 10 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 improvement ranges from 8% to 13% in the physical testbed and from 2% to 17% in the virtual one.

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 three SequenceCount jobs to the clusters at the same time. The container scheduler is the default Capacity Scheduler in YARN. Fig. 11 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%". The reason may be that PGSS has the chance of pending some 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. Different scheduling algorithms 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 uplinks and downlinks of the low-bandwidth nodes is 512 Mbps and it is 1024 Mbps for the high-bandwidth ones. 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. The order of fetch requests is: one shuffle of the node requests to schedule all its fetch flows, then another shuffle on another node requests to schedule all its fetches, and go on. 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

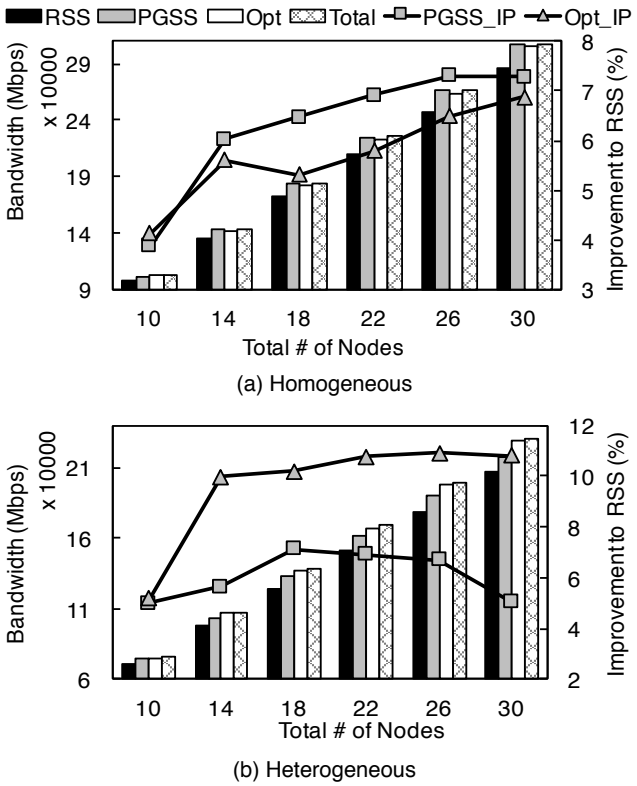


Fig. 12. Bandwidth Utilization and Improvement (Suffixed with IP) of RSS, PGSS and the Online Optimal Algorithm (Opt) with Different Amount of Nodes in the Homogeneous Settings and the Heterogeneous Settings

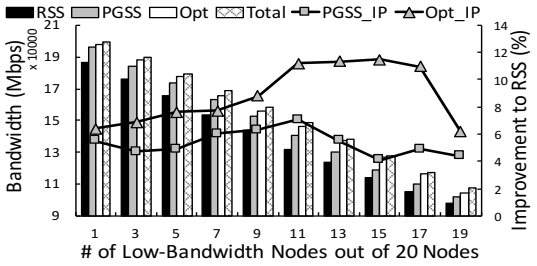


Fig. 13. Bandwidth Utilization and Improvement (Suffixed with IP) of RSS, PGSS and the Online Optimal Algorithm (Opt) with Different Degrees of Network Heterogeneity

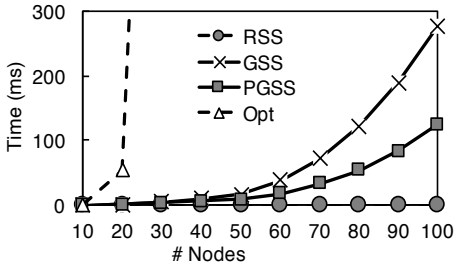


Fig. 14. Scheduling Overhead Time per Flow of the RSS, GSS, PGSS and Online Optimal (Opt) Algorithms with Different Number of Nodes in the Cluster

on the same node to start to fetch data from all the other nodes again if it has free fetchers.

All tests about PGSS and the online optimal algorithm are ran 10 times, and it is 100 times for tests about RSS due to its property of total randomness. The bandwidth utilization value is obtained as the mean value of all runs. Simulation results show that PGSS performs much better than RSS in bandwidth utilization with different numbers of nodes and 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 ones. Fig. 12 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 the scheduler have already seen. 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. 13, where the left and the right ends of the horizontal axis stand for a higher degree in homogeneity and the middle part of the figure stands for a 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 is about 7% in the simulation. It show the benefits of the bandwidth-aware source selecting algorithm in the heterogeneous networks.

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 number of nodes grows) in the cluster. Fig. 14 depicts the overhead time of RSS, GSS, PGSS and 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 distributed resource managing platforms, like YARN [5] and Mesos [9], 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 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 the network performance and the fair share of network bandwidth in datacenter networks at the network level, either by individual flow scheduling [12], [19], [20], [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 better understands the shuffle phase by holding the whole picture of the process and schedules the sources of TCP flows at the application level. BAShuffler estimates the TCP network but leaves the policy of sharing bandwidth to TCP's natural behaviors. Distributed platforms 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 decreases 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, each TCP flow often does not have an equal share of the bandwidth in the network, especially in heterogeneous network environments. The throughput of a shuffle task cannot 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.

6 Conclusion

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

Acknowledgments

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

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*, pp. 1383–1394, NY, USA, 2015.

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

[6] Y. Chen et al., "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of VLDB*, vol. 5, no. 12, pp. 1802–1813, 2012.

[7] 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.

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

[9] 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.

[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.

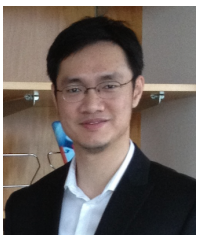
[14] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the ACM Conference on SIGCOMM*, New York, NY, USA, 2014, pp. 443–454.

[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] F. Liang and F. C. M. Lau, "Bashuffler: Maximizing network bandwidth utilization in the shuffle of yarn," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 281–284.
[17] 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.
[18] 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.
[19] 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.
[20] L. Popa et al., "Faircloud: sharing the network in cloud computing," in *Proceedings of the ACM Conference on SIGCOMM*, 2012, pp. 187–198.
[21] 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.
[22] 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.
[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.
[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] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. cambridge university press, 2005.

Feng Liang received the BS degree in software engineering from Nanjing University in 2012, and the PhD degree in computer science from The University of Hong Kong. His research interests are mainly on distributed file systems, distributed computing, formal methods for distributed systems 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