Fine-Grain Resource Allocation and Semantics-Aware Data Partitioning in Distributed Computing

by

Liang Feng

梁鋒

Department of Computer Science

The University of Hong Kong

Supervised by

Prof. Francis C.M. Lau

A thesis submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at The University of Hong Kong

Nov, 2016

Declaration of Authorship

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualications.

Signed:

Liang Feng
Nov, 2016

Abstract of thesis entitled

"Fine-Grain Resource Allocation and Semantics-Aware Data Partitioning in Distributed Computing"

Submitted by

Liang Feng

for the degree of Doctor of Philosophy
at The University of Hong Kong
in Nov, 2016

Existing popular distributed computing frameworks provide robust and highly-available platforms for high-performance computing on a cluster. Notably, MapReduce is highly effective in batch processing of exceptionally huge volume data in a large cluster; and Spark, known for its flexible distributed programming model, provides faster distributed data processing if the data can fit into the memory of the cluster. However, most of these frameworks fall short of offering a satisfactory solution to the problem of underutilization of the cluster's resources, and they are prone to suffering from unnecessary excessive workloads that are due to their ignorance of applications' semantics.

This thesis presents solid solutions—SMapReduce, BAShuffler and Confluence—to improve the performance of distributed computing from different angles from the framework to the application. We study thoroughly the reasons for underutilization of both computational resources and network resources in distributed platforms. SMapReduce maximizes the MapReduce computational throughput by dynamically managing the concurrency level of the computer nodes based on the thrashing phenomenon. BAShuffler utilizes the TCP's max-min fairness bandwidth estimation to schedule the MapReduce-like shuffle flows so that the

network bandwidth in the cluster can be fully utilized. These two solutions operate at the level of fine-grained cluster resources allocation. We then propose an application-semantics-aware data partitioning scheme, Confluence, which partitions the data based on key dependency relationships at the application level. Confluence can reduce the network traffic workloads in iterative distributed computing models. We implement all three proposed solutions in typical pervasive distributed platforms such as Hadoop and Spark, and evaluate them using various realistic benchmarks in a moderate-size testbed. The evaluation results show that these solutions can improve the performance of applications significantly.

[**272 Words**]

Acknowledgements

It is my pleasure to acknowledge the people who made this thesis possible.

First of all, I thank my parents, who are always the strongest backing all through my life.

I am grateful to my supervisor, Professor Francis C.M. Lau, the best tutor, both of research and of life, I have ever met. With his encouragement to pursue the science, I enjoy the freedom to see and solve the problems independently not only in studying the field of computer science, but also, in a broader term, in seeing the world.

I would like to thank my officemates, in HW324A and MB112, with whom the life in HKU is so colorful, warm in heart, and comfortable. I will never forget the time with them.

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

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

BSP	**B**ulk **S**ynchronous **P**arallel
MMF	**M**ax-**M**in **F**airness
GSS	**G**reedy **S**ource **S**cheduling
PGSS	**P**artially **G**reedy **S**ource **S**cheduling
RSS	**R**andom **S**ource **S**cheduling
CKP	**C**onfluence **K**ey **P**artitioning
RKP	**R**andom **K**ey **P**artitioning
KDG	**K**ey **D**ependency **G**raph
DAG	**D**irected **A**cyclic **G**raph

Dedicated to my parents.

Chapter 1

Introduction

1.1 Motivation

These years, the rapid development of distributed computing has been greatly facilitated by various reliable, fault-tolerant and high-throughput distributed resource managing platforms and the corresponding programming frameworks [29, 40, 42, 57, 59, 60, 66, 76, 84, 87, 88]. These platforms offer the methods to control the computational resources of the cluster and to divide and run the computing jobs with allocated resources. However, they are far from reaching the goal of fully utilizing the resources in the cluster so that the job throughput is maximized and the goal of optimizing the programming framework so that the minimal resources are required.

First, in the pervasive distributed platforms, such as Hadoop [81] and Mesos [40], the amount of resources (mainly memories and CPU cores) that are exclusive to the task is statically self-defined by the users, rather than by the platform system, and the task will monopolize this amount of resources all along its lifetime. This resource allocation mechanism hints the following potential problems of under-utilizing the cluster resources: 1) the proper amount of resources for the optimal throughput performance is usually hard to figure out and it heavily relies on the experience of the users as well as requires lots of parameter tuning work; 2) as

the amount of resources occupied the task is no lower than the peak resource requirement of the task and is fixed during the task lifetime, the redundant resources during the idle period cannot be used by any task and are thus wasted. A dynamic method that allocates the proper amount of resources for a task based on the runtime performance information can help to better utilize the cluster resources.

Second, when referring to the resources in the distributed platforms, they usually indicate only the memories and the CPU cores. However, the network bandwidth is a critical performance factor for the communication-heavy tasks, such as the join operation of Hive [76] and the shuffle phase of MapReduce [29] and Spark [87]. Without the control on the utilization of network bandwidth, the network communication becomes the bottleneck of the job, especially for the bulk synchronous parallel (BSP) [20] jobs, where the computing tasks of the latter stage wait to start until all the network communication workloads of the previous stage finish. However, as network bandwidth is the shared resources between different nodes in the cluster, the traditional control mechanism on the node-monopoly resource, e.g., wrapping the memories and CPU cores in the node as the resource container for the task execution, is not applicable to the network bandwidth.

Third, when assigning the application tasks to the computational resources in the existing distributed platforms, the system is unaware of the semantics of the applications. In the iterative distributed computing model such as Spark, where the key-value data entries need to be partitioned to dedicated locations (computer nodes) between any two successive computing iterations, the ignorance of the application semantics can add unnecessary network traffic workloads to the system between iterations. With a better understanding of the data dependency relationship across iterations, the related data of different iterations can be co-located by a global key partitioning scheme to reduce the network traffic.

The above discussions cover different aspects of the problems of distributed computing from the framework to the application. This thesis introduces three solutions to solve each of these problems.

To improve the underutilization of the cluster computational resources, we implement SMapReduce [52] based on Hadoop to dynamically control the concurrency level in the nodes so that computational throughput in each node is maximized. With the observation of the multi-thread thrashing phenomenon [31] and the thorough analysis on the factors that determines the execution time of the distributed jobs like MapReduce, SMapReduce manages the working slots based on the runtime performance information of the distributed jobs. SMapReduce offers significant performance improvement for the compute-intensive jobs.

To further refine the control on the network bandwidth in the distributed platforms, BAShuffler [53] is proposed in this thesis to schedule the source nodes of the communication flows in the shuffle phases (where data distributed in a set of nodes are to rearrange their locations for the next-stage computation) of the distributed operations. BAShuffler applies the notion of max-min fairness (MMF) in TCP communication to make the precise estimation on the network bandwidth allocation status and uses the algorithm called Partially Greedy Source Selection to select the flows so as to maximize the cluster network bandwidth utilization. In contrast to SMapReduce, which focuses on the compute-intensive jobs, BAShuffler can greatly accelerate the communication-intensive jobs.

Key dependency graph (KDG) is proposed as the novel method to depict the key dependency relationship between data of different distributed computing iterations. Confluence Key Partitioning (CKP) relates the distributed system partitioning scheme with the application semantics, which is available to system via the new KDG programming interface. By partitioning the KDG-related data entries in the same node during several successive computing iterations, the data transfer workloads between nodes in the shuffle phase can be avoided. In some applications such as the matrix multiplication algorithm, the data transfer size between iterations can be even eliminated to zero. Confluence is proven in Spark to decrease the communication cost of various typical applications with predictable sizes while not introducing extra imbalanced workloads.

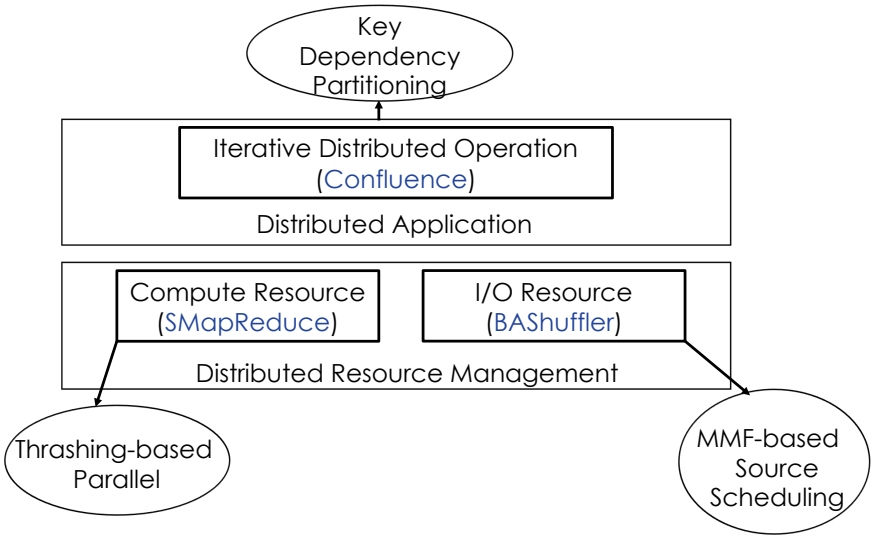


FIGURE 1.1: The Positions of the Three Solutions in the Picture of Distributed Computing and the Corresponding Major Techniques

This thesis introduces the work to improve the performance of distributed systems in the progressive way. Fig. 1.1 portrays the positions of these three solutions in the structure of distributed computing and the key techniques used, respectively. SMapReduce and BAShuffler provide finer-grained control on the computational resources and the network bandwidth resources, respectively, to achieve higher resource utilization and job throughput, while Confluence reduces the communication workloads regardless of the resources allocation policy of the underlying platforms. The three pieces of work enhance different aspects of the distributed systems independently, but they can also work together seamlessly to provide further potential improvement.

1.2 Contribution

The contributions of the work in this thesis are listed as follows:

- This thesis models the execution time of the MapReduce jobs with respect to the job type of map-heavy or reduce heavy, the synchronization barrier of the map and reduce phase, and the concurrency level of the tasks in the nodes. This model distinguishes from others for its recognition of the MapReduce job types and synchronization barrier, which enables the distributed platforms (e.g, Hadoop) to understand more the nature of the jobs based on the runtime statistics and performance. Under the guide of this model, in order to gain higher throughput and lower execution time, the distributed platforms can use different resource allocating policies for different types of jobs and dynamically manage the resources for the job when it is in different working phases.

- SMapReduce is implemented based on Hadoop to dynamically control the concurrency levels in the computer nodes so that the maximum throughput is achieved. SMapReduce manages the number of working slots based on the performance model and the runtime statistics, detects the peak throughput of the node by the observation of the "thrashing" phenomenon, and verifies its advantage over the static resource allocation policy of the original Hadoop on diverse typical benchmarks. SMapReduce can generate 1.4x higher job throughput in the map-heavy benchmarks.

- The analysis of the flow source selection policy reveals the cause of the underutilization of the network bandwidths in the shuffle phase of the distributed operations. This thesis proposes the method to estimate the cluster network bandwidth allocation by the max-min fairness phenomenon provided the information of the link bandwidth capacities and the sources/destinations of the flows. The max-min fairness estimation offers the simple yet reliable method to monitor the network bandwidth utilization status.

- BAShuffler is introduced to greatly improve the network performance for the shuffle-heavy distributed jobs. Without touching the underlying network protocol, BAShuffler improves the bandwidth utilization by scheduling the selection of flow sources at the application level. BAShuffler uses the cost-efficient PGSS algorithm to gain the maximum max-min fairness

bandwidth utilization. It is embedded in Hadoop and can easily run on the clusters of diverse network environment, no matter physical or virtual, with homogeneous link capacities or heterogeneous.

- This thesis also first points out the problem of repeated re-partitioning of key-value data between different iterations in the iterative distributed operations, due to the unawareness of the application semantics of the keys. The unnecessary re-partitioning increases the network traffic workloads for the cluster. Confluence is proposes the solve this problem by constructing the application semantics as key dependency graphs and can avoid the re-partitioning network workloads with guaranteed skew performance. The use of Confluence is illustrated in several typical iterative distributed applications with promised network traffic reduction.

1.3 Thesis Organization

The rest of the thesis is organized as follows. The related work on improving distributed computing is introduced in Chapter 2. Chapter 3, 4 and 5 present in details the rationalities, designs and implementations and evaluations of SMapReduce, BAShuffler and Confluence, respectively. Chapter 6 concludes this thesis.

Chapter 2

Related Work

The related work is introduced in the categories of distributed systems and network, distributed scheduling and distributed models and applications.

The part on systems and network is related to the existing distributed systems for different use cases and the required network environment for the data centers. They affect the art of design of SMapReduce and BAShuffler, which aim to produce higher-performance distributed systems. The part on distributed scheduling mainly introduces the work on the distributed task scheduling and data scheduling in the distributed systems, and the part on the models and applications introduces the popular distributed computing models and the corresponding real-life applications. These two parts guided the designs of all of the three solutions, SMapReduce, BAShuffler, and Confluence. The work on improving the distributed systems need fully understand the models and applications run on top of the systems.

2.1 Distributed Systems and Network

2.1.1 Distributed Resource Management

Existing resource managing platforms for clusters, like YARN [77] and Mesos [40], allow distributed computing frameworks run on the cluster with fine-grain control of computation and storage resources. Comparing to the static slot-based implementation of HadoopV1, tasks run within the resource containers and make better utilization of the cluster resource. However, different configurations of resources for each container can yield performance that varies greatly and the best resource configuration is generally unpredictable. YARN also assigns map tasks at a higher priority than the reduce tasks to prevent too many reduce tasks blocking the process of the map tasks. But unlike the dynamic allocation decision in SMapReduce, the priority design is simple and does not consider the runtime performance to determine an optimal allocation policy in order to gain a higher job throughput. Moreover, they do not have control over network resources in the cluster for tasks that produce a fair amount of traffic. 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.

2.1.2 Datacenter Network

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 [13, 36, 65, 74] or by scheduling the flows in terms of "coflow" [24, 26, 32, 67, 91]. In contrast, 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 behaviors. YARN with BAShuffler can easily be ported to any commercial cluster without changing the underlying network.

Orchestra [25] 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.

Specifically, Sylvain et al. [35] 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.

2.2 Distributed Scheduling

2.2.1 Distributed Task Scheduling

Many researchers have also tried to optimize the task scheduling policy for different application scenarios. They design schedulers to serve different goals, including finishing the jobs before the deadlines [64] [78], maintaining the fairness in the shared environment [43, 86], and allowing priority in jobs [70].

Some have developed scheduling algorithms in specific hardware environments, such as the hybrid computing environment [54] and the heterogeneous cluster environment [89]. Instead, SMapReduce aims at achieving the optimal job throughput in the homogeneous cluster environment. It tries to reach this design goal by considering how many slots should be allocated for the tasks dynamically, so that the resources in the system are fully utilized.

ShuffleWatcher [7] schedules the locality of the map tasks and the reduce tasks so that the shuffle traffic across nodes and racks can be decreased in the multi-tenant cluster environment. It also delays the shuffle tasks when the network traffic is heavy. The shuffle scheduling is roughly in the granular of tasks instead of flows, and it does not consider maximizing the bandwidth utilization based on the bandwidth allocation status as BAShuffler does.

2.2.2 Data locality

The distributed computing paradigms like MapReduce adopt the data locality principle, either to place the computing tasks closed to data locations in priority to reduce the traffic workloads of data migration [29, 89] or to avoid the unbalanced allocation of workloads by considering the compute skew problems [50, 51]. They are concerned with the placing of map tasks where the required input data are self-contained. As the input data of each shuffle task are distributed across the cluster, such task-to-data approach cannot help in the shuffle tasks. Confluence localizes the data and tasks by partitioning the data corresponding to each pure confluence subgraph with multiple shuffle iterations in the same node, which eliminates the shuffle traffic of the following iterations while not impacting the balance of workloads.

2.2.3 DAG scheduling

Similar to the DAG task scheduling algorithms [5, 42, 68, 90] that schedule the tasks to based on the task dependency graph, Confluence optimizes the partition location based on the key dependency graph. The difference is that the DAG task scheduling algorithms try to parallel the execution of jobs that are not depended on each other, while Confluence isolates the partitions of the data with dependency relationship in multiple shuffle iterations.

2.2.4 Online Scheduling

Compared to other online scheduling algorithms [82], 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.

2.3 Distributed Models and Applications

2.3.1 Improving MapReduce Model

Breaking the synchronisation barriers between the map, shuffle, sort, and reduce phases can improve the potential parallelism level of the system [28, 38]. Wang et al. [80] proposed a total ordering method when copying outputs of the map for reducing. Total ordering enables parallel execution of shuffle, merge and reduce phases, but requires an extra synchronization between map and shuffle. HPMR [72] introduces pre-fetching and pre-shuffling into MapReduce in the shared environment. However, these solutions only maximize the logical parallelism of different phases, but do not consider the full utilization of available physical resources in the system, and thus are unlikely to have an optimal execution throughput of the jobs. SMapReduce, on the other hand, can balance the resources even with the existence of the synchronization barrier and is optimal in terms of the job throughput.

Other optimizations of MapReduce include implementing the MapReduce interface on some special devices or services or modifying the interface to adapt to some specific computing tasks. Cloud MapReduce [56] is implemented on top of a bunch of Amazon cloud services. Mars [39] implements the MapReduce interface on GPUs. Map-Reduce-Merge [83] adds a merge phase after reducing to enable the join operation for relational data processing. CLOUDLET[41] optimizes MapReduce for the virtual machines. Twister [33] extends the programming model and enhances the runtime for iterative jobs. SMapReduce does

not modify the interface of MapReduce and keeps the optimization for all types of jobs transparent to the user.

2.3.2 Iterative distributed computing

Some work [18, 33, 37] improves the distributed frameworks for the iterative MapReduce-like computing by extending the programming interfaces to support multiple map and reduce phases. The general idea is to cache the output data of each iteration for the next iteration. Spark [87] provides more kinds of operation interfaces and supports a series of transformations on the distributed input datasets in the memory by the concept of resilient distributed dataset. These iterative distributed frameworks try to minimize the I/O overhead by storing the intermediate data of each iteration in the memory, instead of in the disk like Hadoop. However, they do not consider the heavy network workload during the shuffle operations. Inheriting the benefit of fast memory I/O speed, Confluence decreases the shuffle sizes and alleviates the network workload by considering the key dependency of multiple related iterations.

2.3.3 Parallel&Distributed algorithms optimization

Lots of work is conducted on the optimization of parallel&distributed machine learning algorithms [15] and science computing [48, 61], including the matrix multiplication methods [14, 19]. Generally, they focus on optimizing the algorithm itself by decomposing the tasks to increase the task parallelism and to remove the synchronization boundary between the I/O-intensive tasks and compute-intensive tasks. Sarma et al. [71] represent the map-and-reduce communication cost as the number of data entries needed to generate from the map input with a given parallelism factor. The communication cost for a specific distributed application can be decreased by developing the distributed algorithm with the proper parallelism factor. The term of communication cost is slightly different from the shuffle size in this thesis, where the communication cost is the transfer size of data between different reduce workers, while the shuffle size transfer size

of data between different computer nodes. The algorithms with the minimum communication cost can further decrease the shuffle size by considering the key dependency of the datasets in multiple iterations.

Chapter 3

SMapReduce: Optimizing Resource Allocation by Managing Working Slots at Runtime

3.1 Introduction

Nowadays, big data processing and analysis is critical for many scientific and industrial applications. MapReduce [29], a popular distributed computing framework proposed by Google, because of easy programming, high performance and fault tolerance, is a popular tool for big data analysis [27, 34, 55]. Programmers only need to adapt the computation tasks to the map and reduce interfaces, and MapReduce will take care of managing and running the tasks efficiently in the distributed system.

Hadoop [81] implements MapReduce as an open-source project. Hadoop version 1 (HadoopV1) uses a slot-based design. Each working node (called task tracker) in HadoopV1 executes map tasks and reduce tasks in map slots and reduce

slots, respectively. The numbers of the map slots and reduce slots are configured statically before system startup and cannot be changed at runtime. Map tasks and reduce tasks are assigned to the free slots accordingly. As the workload of many MapReduce jobs can vary greatly during runtime, the simple design and inflexibility of static working slots cannot adapt to the dynamic working behavior, resulting easily in underutilization of available resources or depletion of some resources.

Hadoop evolved to version 2 which is known as YARN [77]. YARN is container-based and treats the resources in the cluster as a combination of memories and CPU cores. The users of YARN can configure the memories and CPU cores of the containers which are used to run the tasks of the job. Thus, YARN offers a more precise control of the resources of the cluster than HadoopV1. However, to determine the suitable amount of resources to assign to the containers is often largely a guesswork by the users. If too little resources is assigned, some tasks might fail due to the lack of memories at runtime; if too much is assigned, a few containers would fill a working node, and the allocated resources could end up poorly utilized. In practice, users tend to configure the container resources lavishly, and thus underutilization of resources is a common phenomenon, which means that optimal throughput is hardly achieved.

HadoopV1 and YARN use different methods to allocate the resources to the jobs. But the resource configuration problem in either version can be seen as the same fundamental problem: to find out the *proper* number of concurrent tasks (or working slots) to run on a working node in order to achieve higher job throughput. The challenge is that this number cannot be decided statically in a dynamic environment. Figuring out this proper number of concurrent tasks is not easy as the jobs can be very different in terms of their compute logic and the data I/O size.

Another problem with MapReduce is that there is a synchronized barrier between the map phase and the reduce phase. The outputs of all map tasks need to be stored locally in some intermediate files and the reduce tasks must shuffle all the map outputs of a particular partition of the problem to the working nodes

before these nodes can start reducing the data. This barrier prevents the reduce function from executing in parallel with the map function of the same job which can increase the total execution time of the job if the map outputs are plenty. YARN sets a higher scheduling priority for map tasks than reduce tasks to make sure the maps tasks can obtain more resources than the reduce tasks. But the optimal scheduling policy for MapReduce jobs to achieve a higher job throughput has not been addressed.

In this chapter, we would not try to break this barrier between the map phase and reduce phase. Breaking this barrier either introduces another barrier or requires extra resources to increase logical parallelism, but optimal throughput cannot be guaranteed. Instead, we study the behavior and performance of MapReduce under different resource configurations; we model the job execution time in terms of the resource capacity and resource allocation, and propose a working slot allocation method which can help the system to achieve the maximum parallelism on both sides of the barrier. We develop SMapReduce [52], which can dynamically manage the working slots for map and reduce tasks at runtime to achieve the minimum job execution time, as a result of maximizing the task throughput with aware of the characteristics of the execution model of the MapReduce jobs.

This chapter is organized as follows. Section 3.2 introduces Hadoop and the thrashing phenomenon in multi-threading systems, and gives the motivation for managing the working slots at runtime. Section 3.3 and Section 3.4 describe the design and implementation of the SMapReduce system. Section 3.5 evaluates SMapReduce using various benchmarks. We highlight some possible future work and conclude the chapter in Section 3.6.

3.2 Background and Motivation

In this section, we introduce the architecture and working mechanism of Hadoop. Understanding how MapReduce works, we then discuss our motivation for developing SMapReduce, a working MapReduce system with dynamic slot management. We also discuss the thrashing phenomenon in multi-threading systems,

which could affect the runtime behavior of MapReduce and is used as a guide in the design of the slot management policy.

3.2.1 Hadoop

Overview of MapReduce

MapReduce is one of the major components in Hadoop. MapReduce divides any given computation into two primitives: map and reduce. The map primitive organizes a list of values by their keys and the reduce primitive aggregates the list of values to a single value for each key.

In HadoopV1, the structure of MapReduce consists of two main types of components: a job tracker (also called the master) and several task trackers (also called the worker nodes). The job tracker is the master computer node that manages the running of the whole system. It maintains the runtime information of the whole system, assigns the map and reduce tasks to the task trackers (based on the heartbeat mechanism between the task tracker and the job tracker), and coordinates the running of the MapReduce jobs. The task trackers are the computer nodes that actually work on the tasks. The task trackers run the map and reduce tasks in map and reduce working slots accordingly.

In YARN, the structure is like that of HadoopV1, except that there is not a single central job tracker. The role of managing the processing of a job is taken up by the application master, which is started on a random node when a job is submitted for running. A single resource manager manages the resources of memories and CPU cores of the cluster and is in charge of allocating these resources to the jobs. The role of the task trackers in HadoopV1 is replaced by that of the node managers. Node managers run the tasks in the resource containers. For uniformity, in the following discussions, we use the *slot* to denote the slot in HadoopV1 and the container in YARN, which runs a task in a working node.

By default, when a map task is assigned to the task tracker by the job tracker, the task tracker will find a vacant map slot to launch a new thread that runs a

Java Virtual Machine (JVM); the map task will run on this JVM. The same for the reduce task. The map task can be divided into three (sub-)phases: the map phase, the sort and spill phase, and plus optionally the combine phase. All the phases do mainly computing, for which the allocated CPU time is the decisive factor of their performance. The reduce task consists of three (sub-)phases: the shuffle phase, the sort phase and the reduce phase. The shuffle phase reads a partition of map outputs of all the map tasks from all the task trackers. The network bandwidth of the cluster is the decisive factor of the performance of this phase. The running of the shuffle phase can overlap with the running map tasks, but it will not end (and hence the remaining two phases cannot start) until all the map tasks finish. This is the synchronization barrier between the map tasks and the reduce phase. With a limited number of map slots and reduce slots in the cluster, and given many map tasks and reduce tasks for the job, there will be multiple waves of map tasks and reduce tasks. For the first wave of the reduce tasks, their shuffle phase can overlap with the running of all waves of the map tasks, but it can only proceed to the remaining phases of the reduce tasks after all the map tasks finish.

Discussion on the Working Slots

We focus on the duration during which the map tasks and the shuffle phases of the first wave of reduce tasks are running in parallel. We refer to this as the *"overlapped section"*. We want to maximize the utilization of the required resources for the execution of the tasks in spite of the existence of the synchronization barrier. The required resources are the CPU time slices (for the mapping) and the network bandwidth (for the shuffling). If we can coordinate the progresses of these two types of tasks, thus making a full utilization of the CPU and network resources simultaneously, we can achieve high performance when running MapReduce jobs.

For map-heavy jobs, the volume of output data needed to be shuffled is small comparing to the volume of input data the map tasks need to handle. In this case, when there are still map tasks working (i.e., before the barrier is reached), the reduce slots will spend most of their time waiting for mapped output data to

shuffle. We could allocate more CPU time slices to the map tasks, that is, more map slots, so that the system can finish the map tasks, cross the synchronization barrier faster, and go on to the reduce tasks. For reduce-heavy jobs, the data needed to be shuffle are large. The shuffle phases require both CPU time and network bandwidth to sort and transfer the data for reducing. If there are too many map slots, besides the overhead of multi-threading scheduling, the shuffle phases may not be able to finish transferring most of the data needed for the corresponding reduce phases, and will have to spend additional time to shuffle data after the barrier. After the synchronization barrier, when the shuffle phases are still running, there are no more map tasks, the reduce phase is waiting, and the system will spend most of the time transferring data through the network, while the CPU is being left idle.

If the Hadoop system can realize the existence of the map and reduce synchronization barrier and allocate the proper numbers of working slots for the map tasks and the reduce tasks dynamically at runtime, the system could make full utilization of the CPU and network resources available and achieve higher performance. This is the motivation behind SMapReduce, an enhanced MapReduce with dynamic slot management.

3.2.2 Thrashing

Thrashing [31] happens when there are too many threads in the system, and the virtual memory system is in a constant state of paging between the disk and the memory; this causes the CPU utilization to drop drastically. A similar phenomenon regarding CPU throughput can happen in the MapReduce system. When the number of working slots for map tasks and reduce tasks increases, CPU throughput increases; but when the number of working slots reaches a thrashing point, the throughput of the working tasks begins to decrease.

Generally, if we allocate more working slots to the map tasks or reduce tasks, the throughput of the tasks will increase. The increase of the number of working slots, however, is accompanied by a corresponding increase in the thread scheduling

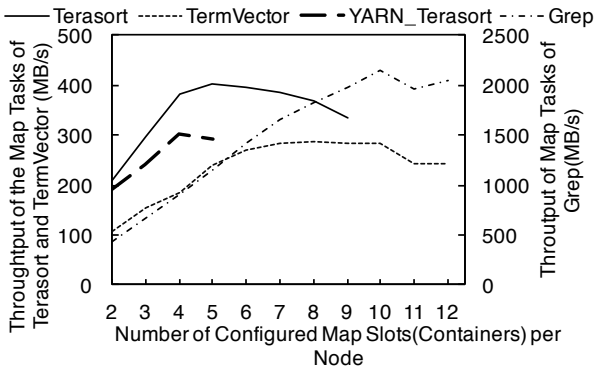


FIGURE 3.1: Thrashing: In the Terasort, TermVector, and Grep benchmarks, the curves of the throughput of the map slots versus the number of map slots in each node begins to fall when the number of map slots reaches the thrashing point.

overhead, which creates a threshold in the number of working slots. When the number reaches this threshold, the scheduling overhead starts to outweigh the benefit gained, and the throughput will begin to decrease with the rising number. Fig. 3.1 shows the thrashing phenomenon in HadoopV1 and YARN with three of the benchmarks we used in our workbench (to be introduced in Section 3.5). We can see from the figure that the map task throughput is sensitive to the number of configured map slots. Finding a proper number of slots for the task is meaningful in order to increase the throughput of the system. In all of the benchmarks, the throughput increases proportionally to the number of map slots. When the number of map slots reaches the thrashing point, the throughput stops increasing or begins to fall. The thrashing points of different job types can be different. In general, map-heavy jobs have a higher thrashing point than reduce-heavy jobs. This is because reduce-heavy jobs spend more resources on shuffling and reducing than map-heavy jobs and suffer an early map thrashing point.

The thrashing point will guide us to decide on the number of working slots to allocate to the map tasks or reduce tasks when we manage the working slots in SMapReduce at runtime.

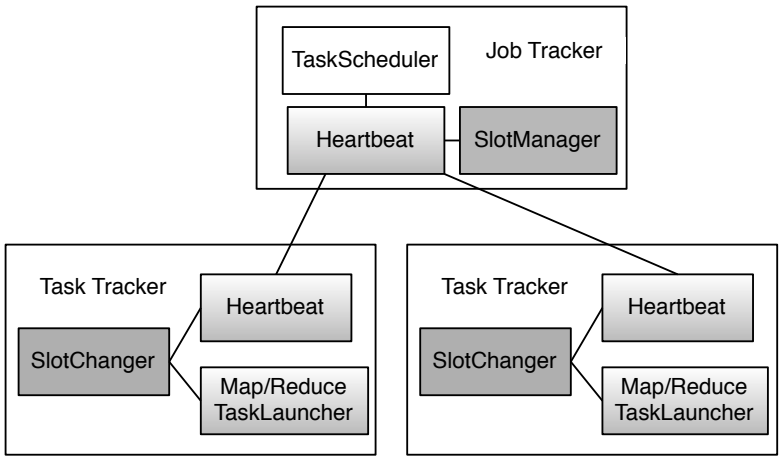


FIGURE 3.2: The Architecture of SMapReduce

3.3 Design of SMapReduce

In this section, we present the architecture of SMapReduce and the design of the working slot management algorithm, and discuss several important issues that affect the design.

3.3.1 Overview of SMapReduce

We design and implement SMapReduce based on the existing framework of the slot-based design of HadoopV1 by adding a working slot managing mechanism. Fig. 3.2 shows the architecture of SMapReduce. The blocks in white color are components that are the same as in MapReduce, the blocks in gradient gray are components that are modified from HadoopV1, and the blocks in dark gray are components that are new in SMapReduce.

The job tracker of SMapReduce consists of three main components: the task scheduler, the heartbeat handler, and the slot manager. The task scheduler decides which task should be run on which task tracker. It is the same as that in MapReduce. The heartbeat handler collects heartbeat information from the task trackers and responds with the commands for the task trackers to execute.

The commands from the job tracker to the task trackers include the tasks to run or clean up. In SMapReduce, job tracker also sends commands of changing the working slot number to the task trackers by heartbeats. The slot manager is a thread that makes decisions on how many slots should be in the task tracker in order to optimize the resource utilization.

The task tracker of SMapReduce is made up of these components: the heartbeat sender, the slot changer and the map task launcher and reduce task launcher. In addition to the task tracker's status and the running status of tasks, the task trackers also supply statistics of the running tasks to the job tracker by heartbeats. These task statistics help the slot manager in the job tracker to determine the types of the running jobs and the numbers of slots needed for both map tasks and reduce tasks. The slot changer changes the number of working slots in the task tracker. the task launchers launch map tasks and reduce tasks on the available working slots.

3.3.2 Slot Manager

The slot manager is the critical component in managing the working slots in SMapReduce. The slot manager makes use the runtime information of the MapReduce jobs to determine the proper map slots and reduce slots for the jobs at that moment.

Balancing between Map and Shuffle Throughput
One important aim of SMapReduce is to properly allocate the working slots for the map tasks and reduce tasks in the overlapped section, so that the jobs can progress to the barrier faster and thus have a shorter execution time.

Based on the workload, the total progress of a MapReduce job can be divided into two parts: the front stretch is from the start of the job to the end of execution of the shuffle phases of the first wave, and the tail stretch is from this point to the end of execution of the job. A job is so divided because in the front stretch, the map phases and the shuffling phases of the same job are running in parallel, while in the tail stretch, only reduce tasks are running.

For the front stretch, SMapReduce tries to balance the allocation of working slots for the map tasks and reduce tasks so that the execution time of this part is minimized. For map-heavy jobs, the map output data are small and it should be easy for the shuffle rate to match the map output rate. If they do match, when the map tasks finish, the shuffle phase is also more or less finished. In this case, the time needed to finish the work of the front stretch of the progress is:

$$t = \frac{M}{T_m},$$

where M is the workload of the map tasks and T_m is the throughput of the map tasks. When the shuffle rate can match the map output rate, since the map workload M is a constant, to achieve minimum execution time of the front stretch, we only need to maximize the throughput of the map tasks in this part. As the map throughput increases, the map output rate also increases. It may happen that the map output rate increases to a point where the shuffle rate cannot catch up. This in fact would be the situation of reduce-heavy jobs.

For reduce-heavy jobs, the data needed to be shuffled in the front stretch is heavy. It is possible that the shuffle rate cannot catch up with the map output rate—i.e., the shuffle phases still have a lot of data to transfer after all map tasks have finished. In the case when shuffle rate cannot match the map output rate, the time needed to finish the work of the former progress part is:

$$t = \frac{M}{T_m} + \frac{R - \frac{M}{T_m} \times T_{r1}}{T_{r2}},$$

where M is the workload of the map tasks, T_m is the throughput of map tasks, R is the workload of the first wave of the shuffle phases, T_{r1} is the shuffle throughput of the shuffle phases when the map tasks are still running, and T_{r2} is the shuffle throughput after the map tasks finish. Note that T_{r2} is a constant in the system since the system only needs to run reduce tasks and there will not be any resource sharing between the map tasks and the reduce tasks. It can be assumed that total throughput of the system is a constant, no matter how the resources are divided between the map tasks and the reduce tasks, and thus we have $T = T_m + T_{r1}$,

where T is the total throughput of the system. The above equation can be simplified to:

$$t = \frac{R + M}{T_{r2}} - \frac{(T - T_{r2}) \times M}{T_m \times T_{r2}}.$$

Since all the variables except T_m are constant, if we want to reduce the time t, the map task throughput T_m should decrease. This makes sense. When map tasks and shuffle phases work together to the full extent, the system is making full use of the available resources working for the job. If the map tasks progress too quickly and finish before the shuffle phases finish, the system will only use part of the resources for the shuffle phases afterwards.

When the map throughput is reduced to shorten the execution in the reduce-heavy case, the map output rate also falls. It can come to the situation where the shuffle rate can match the map output rate again. The state when the shuffle rate can just catch up with the map output rate is called the *Balanced State*. When the system is in the balanced state, the running map tasks and shuffle phases are making full utilization of the resources of the system, and can achieve the minimum execution time of the front stretch of the progress.

As discussed in Section 3.2, the throughput of the map tasks can be controlled by the number of map slots in the system, as long as the map tasks have not reached the thrashing point. In the rest of this chapter, we assume that the MapReduce jobs consist more tasks than the total slots in the cluster when the job starts, i.e., tasks are competing for the slots to start their progress.

Detecting the Thrashing Point

Increasing the slot number does not always lead to increase in the throughput of the tasks. The slot manager needs to detect the occurrence of thrashing when it tries to control the throughput of the map tasks or reduce tasks by adjusting the number of working slots.

Detecting the thrashing point is easy. Take the map tasks for instance. For every number of map slots, the slot manager records the average processing rate of all the map tasks in the system for that slot number. If the number of map slots increases, the slot manager calculates the average processing rate of the

map tasks running on the new number of map slots and compares it with the previous average processing rate. If the current rate is smaller than the previous one, the slot manager knows that the system has reached a thrashing point and will stop increasing the number of map slots.

Switching Map Slots to Reduce Slots
When the front stretch comes to the end, the number of unfinished map tasks decreases and fewer map slots are needed to run the map tasks. In the tail stretch, there are only reduce tasks running. Under this circumstance, the slot manager can reduce the number of map slots and increase the number of reduce slots appropriately to accelerate the execution the remaining unfinished reduce tasks.

However, we will only increase the reduce slots in the tail stretch when the job shuffle size is small. When the job shuffle size is large, increasing the reduce slots will add a large number of threads copying the map outputs, which can jam the network and thus reduce the shuffle rate on the contrary.

3.3.3 Messages in Heartbeat

The slot manager requires statistics of the running job, such as the shuffle rate, in order to balance the map and reduce throughput, to detect the occurrence of thrashing, and to judge the progress of the job. These statistics need to be collected from the task trackers from time to time. We inherits the heartbeat mechanism between the job tracker and the task trackers of MapReduce to support the running of the slot manager.

In addition to the original heartbeats of MapReduce, the task trackers of SMapReduce adds the following information to each heartbeat message: the map input processing rate, the shuffle rate and the map output rate. The slot manager can aggregate these data from all the task trackers.

In the job tracker's end, the heartbeat handler detects whether the number of map slots or reduce slots of a task tracker is different from the number decided by the slot manager. If it is, the heartbeat handler will send a command to ask

the task tracker to update its number of working slots as the heartbeat response. After receiving the command to update the working slot number, the task tracker passes it to the slot changer to handle the changing of the number of working slots.

3.3.4 Lazy Slot Changing in Slot Changer

The slot changer of the task tracker has two roles: one of changing the number of map slots and the other of changing the number of reduce slots. The slot changer does not change the number of the slots directly. Instead, it sends a signal to the task launcher indicating that the number of working slots has changed. The task launcher will then change the working slot number, but lazily.

There is a map task launcher and a reduce task launcher in the task tracker. They launch map tasks and reduce tasks using the available working slots, respectively. When the task launcher receives a signal from the slot changer, the signal can ask the task launcher either to increase or to decrease the number of the working slots. When it is the increase signal, the task launcher should be able to add some working slots which are ready for launching tasks. When it is the decrease signal, the task launcher may be in a state that all the slots are working on the tasks, which means that there is no free slot at that moment; if the task launcher shuts down one slot immediately, the running task, which is in the middle of its progress, must be terminated and rescheduled in another free slot later. This rescheduling overhead should be avoided because it wastes the resources that have already been allocated to that task. If the slot changing action is frequent, the rescheduling cost can be substantial.

The task launcher therefore applies the lazy policy in changing the number of the working slots. When it needs to reduce the number of the slots and there are not enough idle slots to shut down, it will shut down the idle slots first and remember the number of slots that still need to be shut down when there are idle slots later. When the busy slots finish running the tasks and become idle, the task launcher will know that it is safe to shut down some idle slots if the

total number of slots is greater than expected. In the case that the task launcher wants to increment the number of slots, it will be safe for the task launcher to add the slots immediately.

3.4 Implementation of SMapReduce

This section will present some of the implementation details of SMapReduce. We implement SMapReduce based on the source code of Hadoop, version 1.0.4, a recent stable version. We have modified mainly the JobTracker class, the TaskTracker class and the MapTask class.

3.4.1 Implementation of JobTracker

In the JobTracker class, much of the work is spent on implementing the slot manager. The slot manager is a thread in the job tracker. The slot manager needs to detect whether the system is in a thrashing state and decides on the proper number of the slots periodically. After every time period, the slot manager is almost certain that all the task trackers have updated their statuses in the job trackers since the last period, and thus can make a more accurate judgment on the state of the system.

There are some issues we need to deal with when detecting the thrashing phenomenon and deciding on the proper number of working slots for the MapReduce jobs.

Slow Start

At the beginning of the execution of the job, the data reported from the task trackers may not be substantive enough for the slot manager to base on to make a decision. Such data can lead to wrong decisions, and thus impact the performance. For example, soon after the job has started and yet no map tasks have finished, the shuffle rate of the system is zero while the map output rate is non-zero. This can lead to a wrong conclusion that the shuffle rate cannot match the map output rate, and the job is suspected to be reduce-heavy.

The slot manager adopts the "slow start" approach to avoid this potential problem. The slot manager will only start working after a certain portion of the map tasks have finished executing and reported their running statistics to the job tracker. In SMapReduce, the value of the start threshold is 10% by default. The slot manager will start working after 10% of the total map tasks have finished.

Suspected Thrashing

As discussed in Section 3.3, to detect the occurrence of map thrashing, the slot manager compares the average map processing rates when the number of map slots increases. The map processing rate is measured by the input rate of the map tasks. However, immediately after responding to the slot change command, the map processing rate of the task trackers will drop slightly at first. So it is not advisable that the map processing rate just after a slot change be used for comparison, which will almost always give the result of the occurrence of thrashing. According to our observation, the map processing rate after a slot change will grow gradually to a stable range after some time. Only then should the slot manager begin to consider whether the system has indeed entered map thrashing.

Because of the nondeterministic nature of distributed systems, the slot manager still cannot conclude that the system is approaching the thrashing state once the map processing rate has grown to the stable range and is smaller than what was before incrementing the map slot number. Instead, under this circumstance, the slot manager will mark the current state as suspected of thrashing, and give the system another chance. If the system is detected to be suspected of thrashing continuously, the slot manager can then announce with confidence that the system is in a thrashing state.

The slot manager does not need to consider any reduce thrashing problem, because the number of reduce slots in the system is usually set to a small number, which is not likely to cause thrashing. The number of reduce slots is set low because one reduce slot can generate several threads, to copy the map output data from all the other nodes in the cluster during the shuffle phase. A large number of reduce slots can cause network jam in the cluster.

Deciding the Slot Numbers

Initially, the slot manager has a specific number of map slots and reduce slots as configured by the user, just like in HadoopV1. It gradually adjusts the number of slots based on the running status of the jobs. To decide the proper number of map slots or reduce slots for the job, the slot manager considers the two progress stretches as discussed in Section 3.3.

In the front stretch of the progress, when many map tasks are running in parallel with the first wave of shuffle phases, the slot manager needs to find out whether the shuffle rate can catch up with the map output rate of a partition of the reduce data. The map output rate of a partition of reduce data is the data needed to be shuffled in the first wave of shuffle phases. It is estimated under the assumption that every shuffle phase needs to transfer the same amount of map output data. Therefore, we have

$$R_m = \frac{n}{N} \times R_t,$$

where R_m is the map output rate of a partition of reduce data, n is the number of the running reduce tasks, N is the number of total reduce tasks, and R_t is total map output rate. The balance level of the shuffle phases and the map tasks can be indicated by:

$$f = \frac{R_s}{R_m},$$

where R_s is the shuffle rate. When the system is not in a thrashing state, if the factor f is greater than an upper bound, we say that the shuffle rate can catch up with the map output rate, and this is the map-heavy case. The slot manager will then increment the map slots by 1, linearly for not producing a too large throughput gap. Usually, the maximum number of the map slots feasible in a node will not be too large, e.g., around 4 to 8, and it will not take too much time for the slot manager to reach the "best" number of map slots. If the factor f is smaller than a lower bound, we say that the shuffle rate cannot match the map output rate, and it is the reduce-heavy case. In this case, the slot manager will decrement the map slots. When the factor f is somewhere between the upper bound and the lower bound, the system is considered in the balanced state between shuffle and map, and will do nothing.

The manage of the number of working slots is per-node based. Each node can have different number of workings at the same time, depending on the runtime performance of each node.

At the end of the front stretch and in the tail stretch, when there are only a few or no map tasks, the map slots are reduced and reduce slots can be increased.

3.4.2 Implementation of TaskTracker and MapTask

The main modification to the TaskTracker class is to add the lazy policy of changing the slot number in the task launcher without affecting the normal execution of the tasks. The lazy policy is implemented by the consideration of the expected slot number the task tracker is going to have. Whenever a busy slot is released, the task launcher checks whether the released slot should be shut down to meet the expected slot number. Whenever a task wants to engage a slot, it makes sure the free slot is not the one that should be shut down because there are already as many slots running as expected, and updates the number of working slots.

In the MapTask class, we added some extra statistics needed by the slot manager. For example, the class records the map output data size of the map task upon completion so that the slot manager can use it to calculate the map output rate of the system.

3.5 Evaluation

The evaluation workbench is configured as follows: The computer cluster consists of 18 nodes, each of which is configured with 4 quad-core 2.53GHz Intel CPUs and 32GB DDR3 memory, running the CentOS 2.6 operating system. One node serves as the name node of HDFS [17], and one as the job tracker of SMapReduce; the remaining 16 nodes are the data nodes of HDFS as well as task trackers of SMapReduce. The nodes are connected by switches with 16 GbE ports. We evaluate SMapReduce in a physical cluster instead of in the virtual environment like that of Amazon in order to minimize the affect of the virtual client machine

Table 3.1: Hadoop Settings

Hadoop Property	**Value**
mapred.reduce.parallel.copies	5
mapred.job.shuffle.input.buffer.percent	0.7
mapreduce.task.io.sort.mb	100
mapreduce.job.reduce.slowstart.completedmaps	0.05

scheduling and virtual network. To reduce the impact of the TCP incast problem in the network, the TCP minimum retransmission timeout (RTO_{min}) is modified from 200 ms to 1 ms [22]. Some of the important setting properties of Hadoop are listed as in Table 3.1, the property names match with Hadoop v1.4.0 if the same property change its name in Hadoop v2.*.

We compare the performance of SMapReduce with HadoopV1 and YARN using the benchmarks from "Purdue MapReduce Benchmarks Suite" (PUMA) [8]. This benchmark suite includes various practical jobs and input data from real life. As we mainly want to see the performance of SMapReduce in the map-heavy benchmarks, we filter some, if not all, really reduce-heavy ones and select 7 out of 13. Table 3.2 lists the data sizes and the characteristics of the jobs. Based on the ratio of the data size to shuffle to the data size of the map input, the jobs Grep, Classify, HistogramMovie and HistogramRating are map-heavy jobs, while the other three, InvertedIndex, TermVector and Terasort are reduce-heavy ones. As the recommended reduce task number is 99% of the number of reduce slots in the cluster, and the default reduce slot number in the 16 task trackers is 2, the reduce task number is set to 30 in all the benchmarks. The block size of HDFS is set to 128MB.

We evaluate the performance of SMapReduce in terms of the execution times of different benchmarks, progress speed, the performance with different data input sizes, the performance under different resource configurations, the effects of the detection of thrashing and the slow start policy, and multiple concurrent job workloads. All the experiment results are the average values of the data collected from two trials.

TABLE 3.2: Details of the PUMA benchmarks

Benchmarks .	Shffle /Input(GB)	Description .
Grep	0.001/293	Search a string pattern
Classify	0.071/275	Classify the movie information into one class
HistogramMovie	0.001/275	Calculate of the number of movies of different grades
HistogramRating	0.001/275	Calculate the number of viewers of different rating movies
InvertedIndex	60/293	Inverted index the words against documents
TermVector	62/293	Determine the most frequent words in a set of documents
TeraSort	306/300	Sort the strings

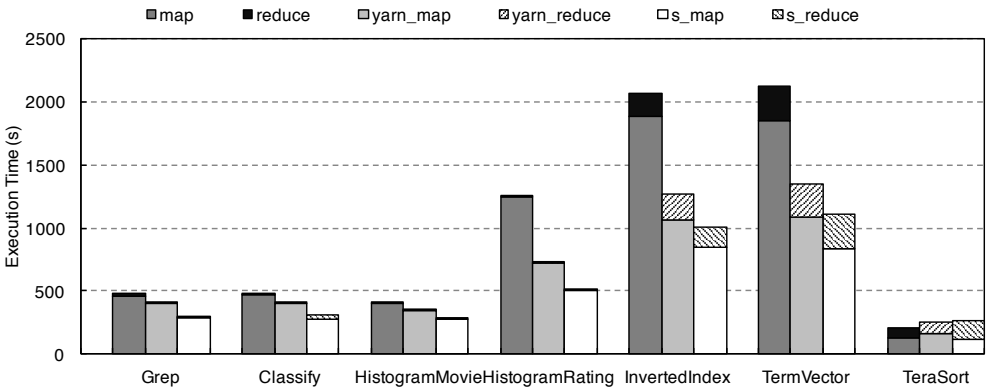


FIGURE 3.3: The Execution Time of Each Benchmark in HadoopV1, YARN and SMapReduce

3.5.1 Performance in Various Benchmarks

To have an overall understanding of the performance of SMapReduce, We run SMapReduce on these benchmarks and compare their execution times in each phase with HadoopV1 and YARN. The configuration of all the jobs in HadoopV1 and SMapReduce is: 3 initial map slots and 2 initial reduce slots in each task tracker. Equivalently, YARN is configured to be able to run 3 map containers and 2 reduce containers concurrently. Fig. 3.3 shows the experiment results. The map time stands for the execution time when map tasks run with the shuffle phases in parallel. The reduce time stands for the execution time after the barrier when only reduce tasks are running.

From the figure, SMapReduce has shorter map time, shorter reduce time and higher job throughput than both HadoopV1 and YARN in almost all benchmarks. Generally, map-heavy jobs and medium reduce-heavy jobs have even higher total performance increase than reduce-heavy jobs. This is because map-heavy jobs usually have a higher thrashing point, and any misconfiguration of map slots and reduce slots leaves plenty of space for optimization. For instance, in the HistogramRating benchmark, In terms of throughput, SMapReduce has 140% and 72% performance increase than HadoopV1 and YARN, respectively. Terasort is the only exception here, where SMapReduce execution time is slightly longer than that of HadoopV1 and YARN. This is because the current slot configuration happens to be optimal for this job, and the management of slots in SMapReduce adds a small overhead that has affected the job throughput. But the overhead is so small that it should be negligible.

The total performance increase is mainly credited with the map throughput increase. Because of the policy of increasing the reduce slots appropriately when map tasks are few or all finished, we can have reduce throughput increase in many benchmarks, such as in InvertedIndex. This policy failed to increase the reduce throughput in some benchmarks, but the effect is very small. In the rest part of this chapter, we will mainly focus on the map-heavy jobs, such as HistogramRating, to explore how SMapReduce performs in the other performance metrics.

3.5.2 Progress Speed

To understand further the effect of slot management on the performance of MapReduce jobs, we record the progress percentage of the finished part of the jobs throughout their execution. The total progress percentage of the job is 200%, 100% for the map tasks and 100% for the reduce tasks. Fig. 3.4 shows the progress percentages over time of the HistogramMovie benchmark running on MapReduce and SMapReduce, respectively. At the beginning, SMapReduce progresses at approximately the same speed as HadoopV1 and YARN. However,

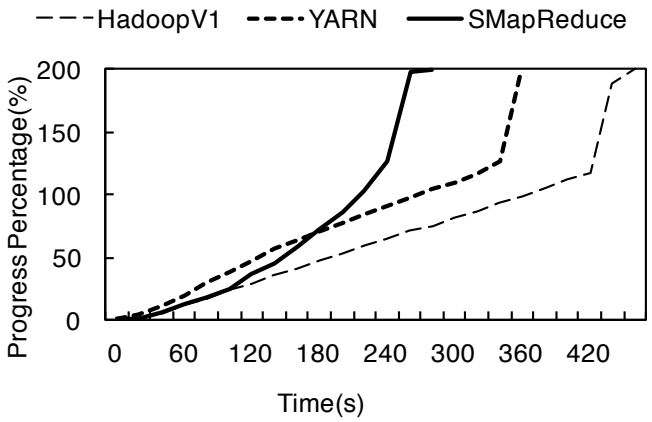


FIGURE 3.4: The Progress Percentage along the Time of the HistogramMovie Benchmark on HadoopV1, YARN and SMapReduce

as time goes by, the slot manager of SMapReduce dynamically adjusts the number of slots for the map tasks so that progress of the job is sped up. As the configuration of slots is getting close to the optimal, the speedup rate increases over time. Without slot management, the job progresses at a constant speed in HadoopV1 and YARN. Note the sharp turns of all the curves at the point slightly above the 100% mark, which is when all map tasks finish.

3.5.3 Different Resource Configurations

In the above evaluations, we configure the number of map slots to 3 and the number of reduce slots to 2. It is possible that HadoopV1 and YARN performs poorer than SMapReduce simply because of the mis-configuration of the number of slots (or container memories in YARN). We evaluate SMapReduce under different configurations of the number of map slots (or containers) and find that SMapReduce still has a shorter map time in most cases. Fig. 3.5 shows the map times of HadoopV1, YARN and SMapReduce under different map slot configurations for the HistogramRating benchmark.

In the HistogramRating benchmark, when the map slot number is between 2 to 6, the map throughput of SMapReduce is 10%-18% higher than that of YARN

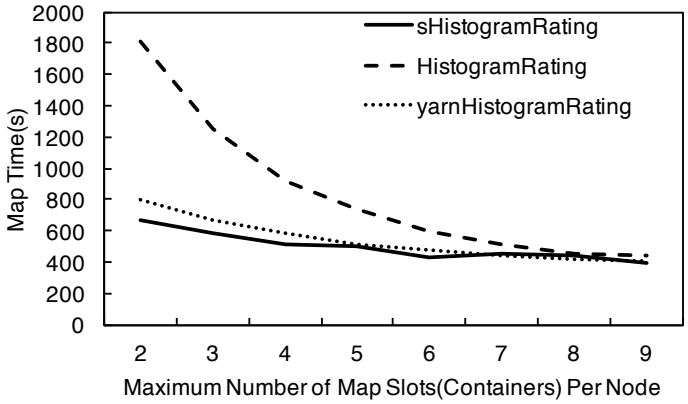


FIGURE 3.5: Map Time of the HistogramRating Benchmark under Different Map Slot Configurations

and 30%-160% higher than that of HadoopV1. For the configuration where HadoopV1 and YARN can achieve the minimum map time—i.e., it happens to be the optimal map slot configuration—SMapReduce can still achieve the same map performance as MapReduce.

We did not measure the performance of SMapReduce vs. HadoopV1&YARN under different reduce slot configurations because the number of reduce slots on each task tracker is usually set to a small number (e.g., 2) to avoid too many concurrent reduce tasks, which can jam the network.

3.5.4 Different Input Sizes

We also evaluate the scalability of SMapReduce. We measure the job throughputs of SMapReduce with input data of different sizes with the HistogramRating benchmark. Fig. 3.6 shows the result. As the input size increases, the job throughput of SMapReduce increases, while those of HadoopV1 and YARN remain almost unchanged. It is because when the input size is large, SMapReduce has more time to adjust the slots to the optimal configuration. When the input size grows to 250GB, the job throughput of SMapReduce is about 1.3 times that of YARN and 2.0 times that of HadoopV1. One conclusion we can make is that

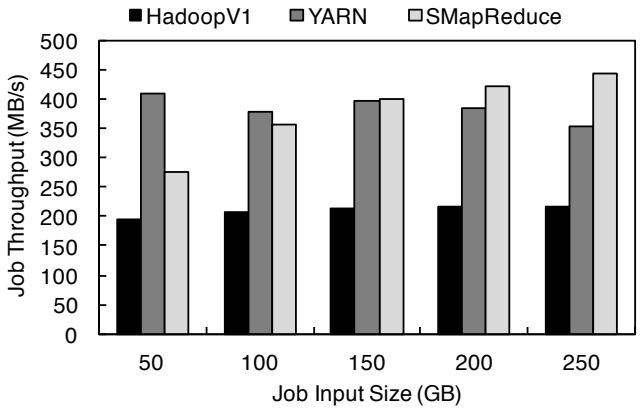


FIGURE 3.6: HistogramRating Job Throughput of HadoopV1, YARN and SMapReduce with Different Input Data Sizes

the larger the input data size, the more benefits we can get from managing the slots at runtime.

3.5.5 Detecting Thrashing and Slow Start

Detecting the occurrence of thrashing and the slow start policy are important considerations in our design of the slot managing algorithm. We run experiments to see the necessity and effects of detecting thrashing and the slow start policy. Fig. 3.7 shows the results of the experiment of comparing the map times of two benchmarks with and without the detecting thrashing mechanism, and with and without the slow start mechanism. Without detecting thrashing, the number of map slots may increase continuously until the map phase finishes. In both benchmarks, without detecting thrashing, the map time of SMapReduce is much longer than that of HadoopV1 and YARN. The mechanism of detecting thrashing can greatly improve the performance of SMapReduce. Without the slow start policy, the map time of the SMapReduce may be either larger or smaller than that of HadoopV1 and YARN, depending on whether the slot manager has made a right decision at the beginning when little runtime information is available. Generally, SMapReduce applying the slow start policy runs the map tasks faster than when without the policy.

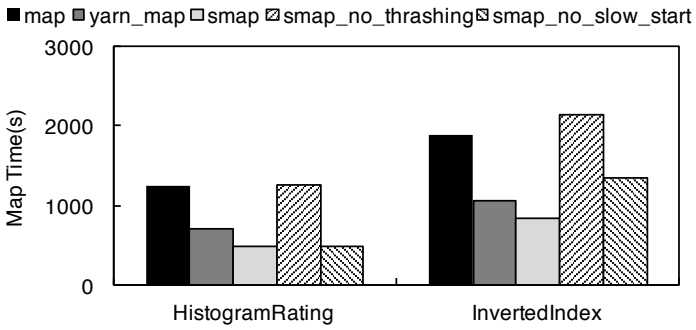


FIGURE 3.7: Map Time with and without Detecting Thrashing and Slow Start Policy

3.5.6 Multiple Concurrent Jobs

Hadoop is a shared environment, and in most use cases, multiple concurrent jobs run in the system. We evaluate SMapReduce with synthetic multiple job workloads. In the multiple job workloads, we submit 4 jobs of the same benchmark in total to the system, and each job is submitted 5 seconds after the previous job. In SMapReduce and HadoopV1, we use the FIFO scheduler for multiple jobs. In YARN, we use the capacity scheduler. Both of them are the default schedulers respectively. The capacity scheduler is similar to the FIFO scheduler, which tries to schedule containers for early submitted jobs first. But the capacity scheduler further considers the map tasks having a higher scheduling priority than the reduce tasks. We compare the mean execution time and the execution time when the last job finishes. Fig. 3.8 and Fig. 3.9 show the performance of the three systems running multiple Grep jobs and multiple InvertedIndex jobs, respectively. SMapReduce has a shorter mean execution and total execution time than HadoopV1 and YARN in both cases. In the Grep workload, for instance, the mean time and the time needed for the last job to finish in SMapReduce are both only about 60% of that in HadoopV1, and only about 70% of that in YARN. SMapReduce clearly outperforms them for multiple concurrent job workloads.

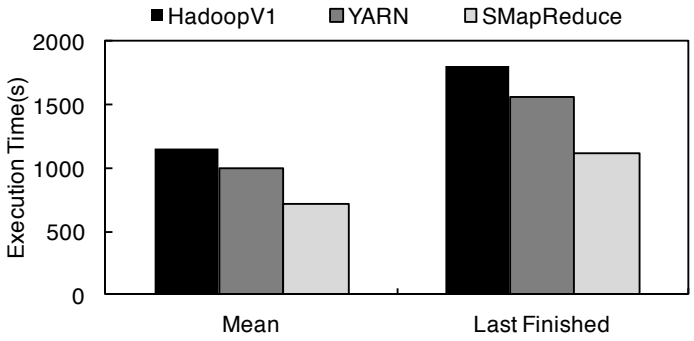


FIGURE 3.8: Mean and Last Finished Execution Time of Multiple Concurrent Job Workload of Grep Jobs

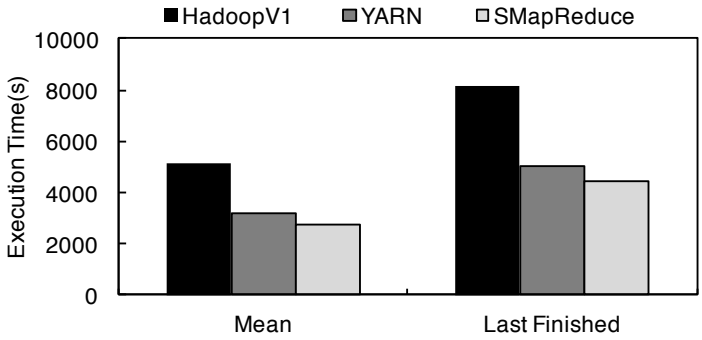


FIGURE 3.9: Mean and Last Finished Execution Time of Multiple Concurrent Job Workload of InvertedIndex Jobs

3.6 Future Work and Conclusion

Currently, SMapReduce only considers the case where the cluster is homogeneous and the data are random in distribution. We are working to extend SMapReduce to the heterogeneous environment, which may be a common setting in some small clusters. Besides, as the newest Hadoop (YARN) has adopted the container-based resource scheduling mechanism instead of the slot-based one, we will figure how the dynamic runtime thrashing-based scheduling can extend to the container capacity scheduling. An further step, we may even try to explore the general mechanism for distributed resource allocation such that the resources are fully

utilized at runtime without the much interference from the users, such as pre-execution resource configurations.

In this chapter, we study how the number of concurrent tasks can affect the performance of MapReduce and build a mathematics model to working out a way to properly allocate the map slots and reduce slots so as to make full utilization of the resources in the system to achieve the minimum execution time of a MapReduce job. We implement SMapReduce, which can dynamically manage the working slots for different types of job at runtime to gain a higher job throughput. Evaluation results show that SMapReduce has remarkable performance improvement comparing to both HadoopV1 and YARN.

Chapter 4

BAShuffler: Maximizing Shuffle Network Bandwidth Utilization by Application-Level Flow Scheduling

4.1 Introduction

The shuffle operation is pervasive in the MapReduce-like [29] distributed operations, such as the "*join*" operation in the distributed database systems [12, 76, 85], the "*reduce*" tasks in the MapReduce systems [29, 77] and the "*aggregateByKey*" operation in Spark [87]. 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 [77] (the latest Hadoop [81]) and Mesos [40] 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 [46].

Unfortunately, unlike memories and CPU cores, which can be encapsulated into containers on a node, the throughput of a communication flow is mainly determined by the available network bandwidth of the network 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, the network bandwidth cannot simply be encapsulated in the 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 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 the situation where some nodes would become 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, whose reason will be analyzed

with examples in Section 4.3.4. A heterogeneous network environment can be quite common in practice for distributed platform users, e.g., YARN users, as physical clusters comprising off-the-shelf machines could be fit with a variety of different network devices [89], or in a virtual cluster, EC2 virtual machines could be running at different network speeds.

Although some work focuses on the network level [24–26, 74], 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. In this paper, our key observation 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 application-level scheduling approach works by making use of both 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). 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 4.2.2.

In this chapter, 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 chapter is organized as follows. In Section 4.2, 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 4.3. Section 4.4 presents the evaluation of BAShuffler. We conclude this chapter in Section 4.5.

4.2 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.

4.2.1 Resource Management and Shuffle in YARN

YARN manages the computing resources in the cluster as a collection of resource containers, each of which consists of memories and CPU cores. The jobs are subdivided 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 has been done to optimize the compute-intensive resource utilization (of CPU, memory, etc.) in Hadoop[30, 52], 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.

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.

4.2.2 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 [36] and fair sharing of network resources [65, 74] 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 [24, 26] 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. Minimizing the completion time of the coflow is NOT the same thing as minimizing the shuffle completion time.

To obtain the optimal scheduling solution that minimizes the shuffle completion time, the scheduler needs to consider both the application level runtime status (all the available map outputs and 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 [69].

Application-level shuffle scheduling has the advantage of knowing the true run-time status of the shuffle. Although it cannot 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.

4.2.3 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 [44]. The MMF of TCP has been extensively analyzed and verified in the literature [16, 23, 47, 79]. 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 [10, 36, 62], it is practical to simplify the datacenter fabric as a non-blocking switch [11, 13, 24–26, 45]. 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 chapter, 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.

4.3 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 data-center 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.

4.3.1 Design of BAShuffler

Design Considerations

In order to improve the shuffle performance of MapReduce in YARN, 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 several factors that can affect the throughput or the execution time of the shuffles. One factor 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 [63]), 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. [81]), 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 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, where different communication patterns can lead to different bandwidth utilization status, which we will illustrate in the coming sections. 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.

Architecture of BAShuffler

The logical position where the BAShuffler components are embeded in YARN is shown in Fig. 4.1. 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.

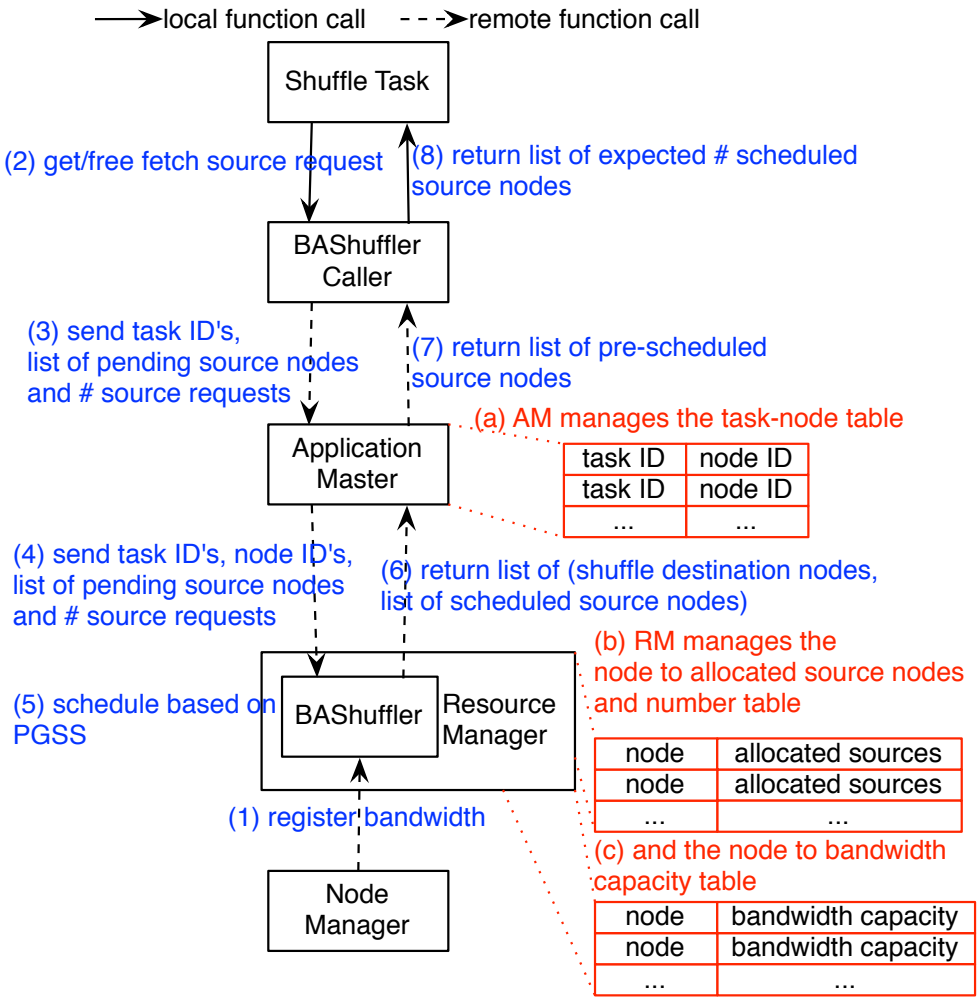


FIGURE 4.1: Architecture Design of BAShuffler

At startup, the node managers register the bandwidth capacities of the uplink and the 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.

4.3.2 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

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*;

```
1  Heaviest ← the heaviest-loaded source nodes in Sources;
2  MaxBandwidth ← 0;
3  foreach Source ∈ Heaviest do
4  |   Util ← the MMF bandwidth utilization of the whole cluster after adding
   |   Source to Pattern;
5  |   if Util > MaxBandwidth then
6  |   |   MaxBandwidth ← Util;
7  |   |   Selected ← Source;
8  |   end
9  end
10 add Selected to Pattern;
11 update the load count of Selected;
```

FIGURE 4.2: Pseudo Code of the Partially Greedy Source Selection Algorithm

a new fetch to schedule, the shuffle selects the pending source node such 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.

4.3.3 Partially Greedy Source Selection

Just like the other online scheduling algorithms [73, 82], 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. 4.2. 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.

Table 4.1: Time Complexity of Different Scheduling Algorithms for Scheduling Each Request

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

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 4.1. 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 4.4 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

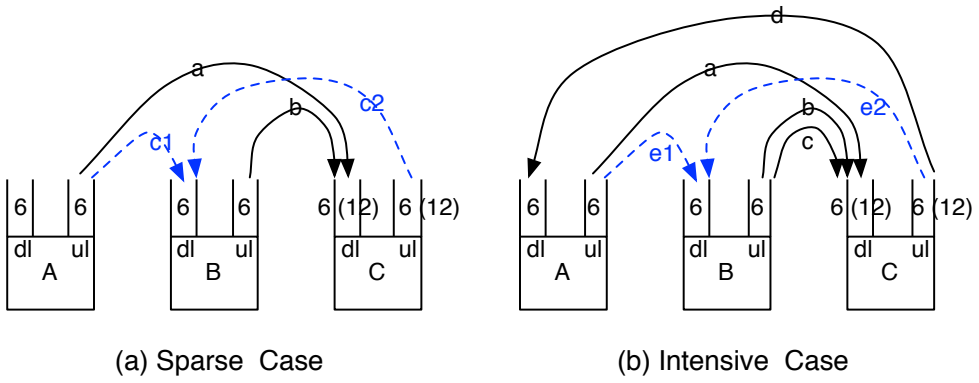


FIGURE 4.3: Scenarios of Selecting the Source in Uneven Flow Pattern

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.

4.3.4 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 discussed in Chapter 4.2.3, 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. 4.3 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,

Table 4.2: 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 4.3: 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

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.

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 4.2 (for the sparse case) and Table 4.3 (for the intensive case), respectively, where "Nil" in an entry means the bandwidth allocation before the source selection. Different selection decisions can lead to different flow bandwidth allocations and overall bandwidth utilizations. In the sparse case (Fig. 4.3), 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.

TABLE 4.4: 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

In the intensive case (Fig. 4.3), 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 [25], 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.

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. 4.3 again. In the heterogeneous network setting, the MMF bandwidth allocation of each flow is shown in Table 4.4 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.

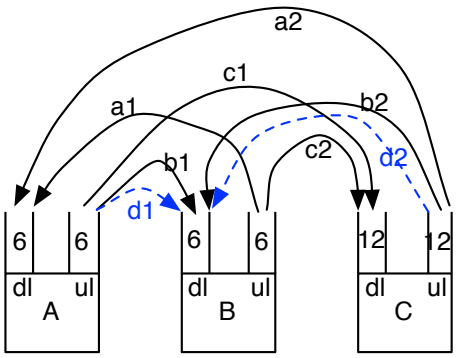


FIGURE 4.4: Selecting the Source in the Even Flow Pattern

TABLE 4.5: 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

Fig. 4.4 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 4.5. 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.

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.

4.3.5 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. 4.1). 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.

4.4 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

Table 4.6: Benchmark Dataset Size (GB)

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

users. Furthermore, we also evaluate the performance of BAShuffler in clusters of different scales and degrees of heterogeneity by simulation.

4.4.1 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 [2] having a heterogeneous network.

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 $\times$ 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 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 [8]. 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 4.6.

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.

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. 4.5 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.

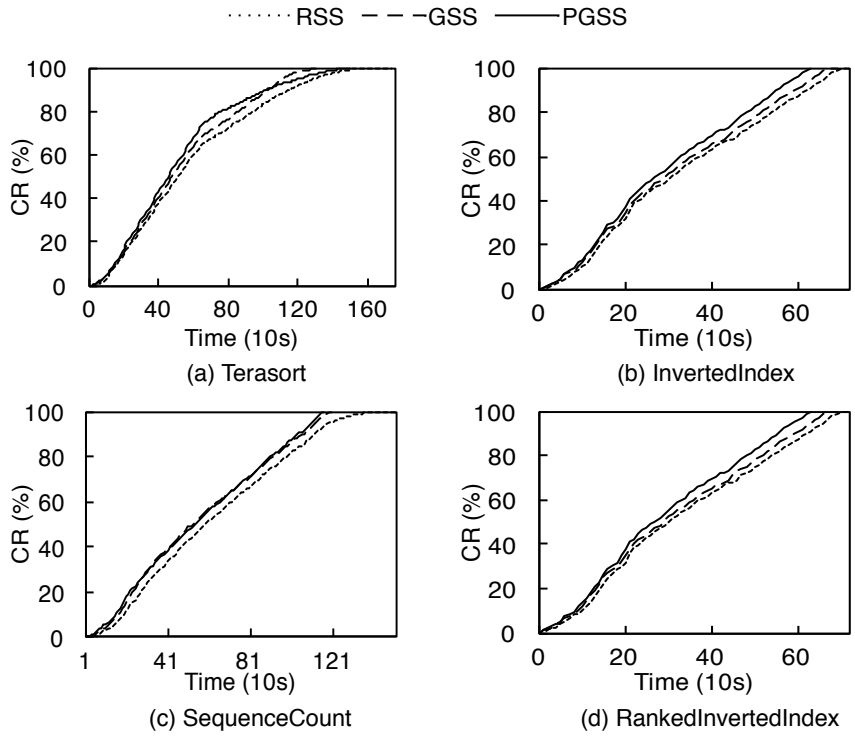


FIGURE 4.5: Cumulative Completion Ratio (CR) of the Overall Shuffle Workload of RSS, GSS and PGSS in Various Benchmarks along the Time in the Physical Cluster

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.

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. 4.6. GSS and PGSS gain relatively high shuffle rate speedups as compared to RSS in all benchmarks, which are as high

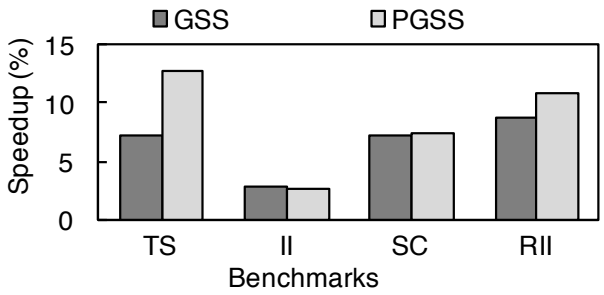


FIGURE 4.6: GSS and PGSS Speedup of Average Shuffle Rate of Each Shuffle Task in comparison to RSS in Benchmarks Terasort (TS), InvertedIndex(II), SequenceCount (SC) and RankedInvertedIndex (RII) in the Physical Cluster

as 7% and 12% in the Terasort benchmark. The speedups of PGSS are a bit higher than GSS, e.g., in Terasort and RankedInvertedIndex.

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. 4.7. 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. 4.8 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,

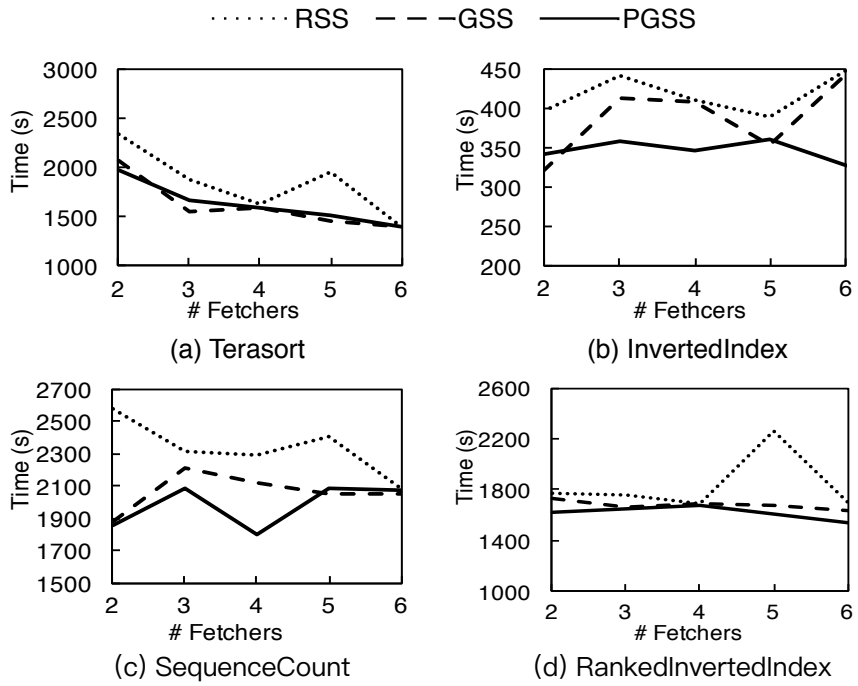


FIGURE 4.7: Reduce Completion Time of RSS, GSS and PGSS in Various Benchmarks with Different Numbers of Fetchers in Each Shuffle Task in the Physical Clusters

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 of RSS 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. 4.8 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.

4.4.2 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 [1]. 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.

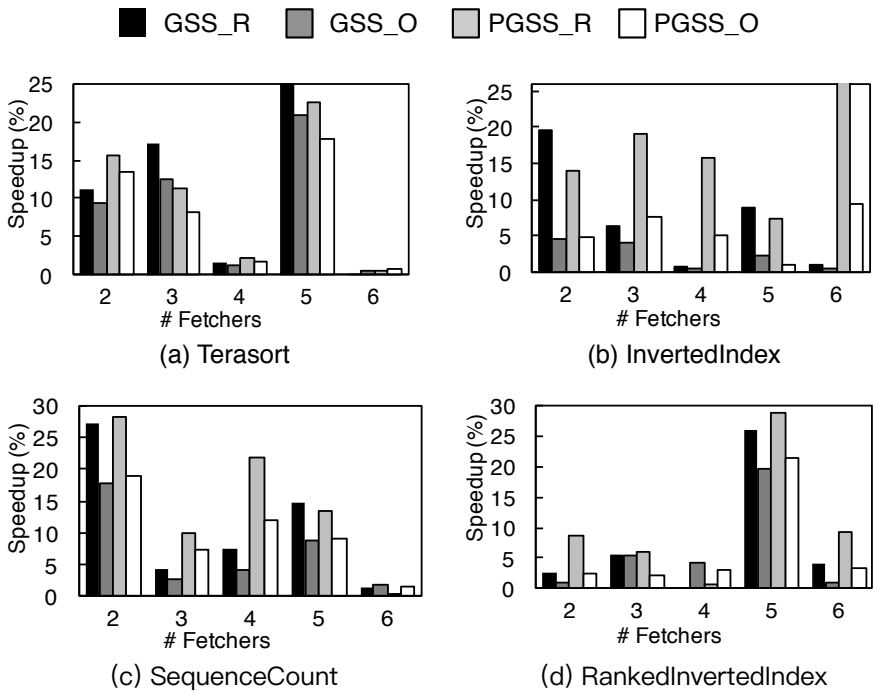


FIGURE 4.8: Reduce Completion Time Speedup (Suffixed with _R) and Job Overall Completion time Speedup (Suffixed with _O) of GSS and PGSS as Compared to RSS in Various Benchmarks with Different Fetcher Numbers in the Physical Clusters

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.

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. 4.9 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

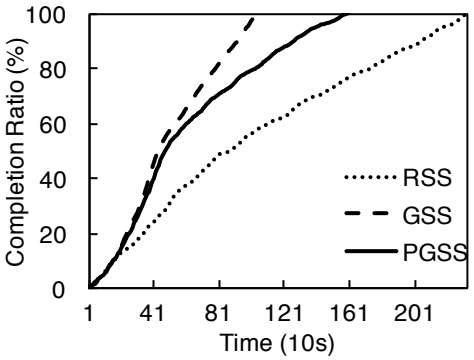


FIGURE 4.9: Cumulative Completion Ratio of Shuffle Workload in the SequenceCount Benchmark along the Time in the Virtual Cluster

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 4.4.3, 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.

Completion Time

The reduce completion of the SequenceCount benchmark with different fetcher numbers in the virtual cluster is shown in Fig. 4.10. Similar to the situation of

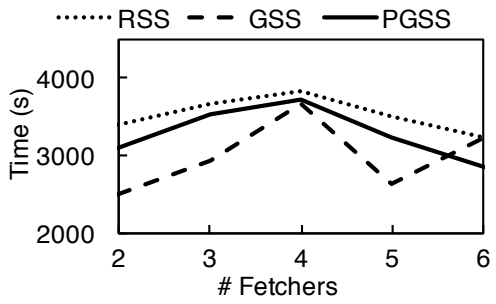


FIGURE 4.10: Reduce Completion Time of the SequenceCount Benchmark with Different Fetcher Numbers in the Virtual Cluster

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.

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. 4.11 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.

Multiple Concurrent Jobs

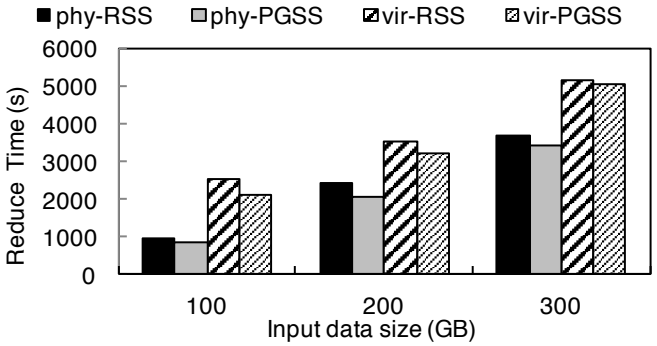


FIGURE 4.11: Reduce Completion Time of SequenceCount with Different Input Data Sizes in the Physical and Virtual Testbeds

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. 4.12 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.

4.4.3 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

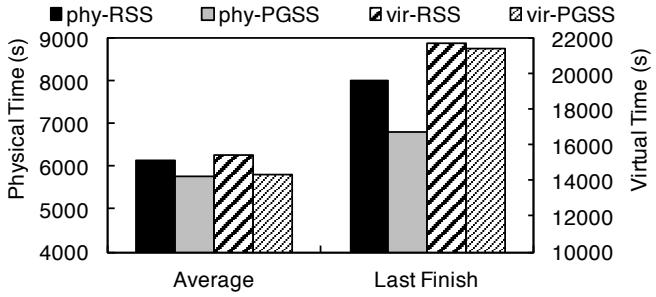


FIGURE 4.12: Job Completion Time of Multiple Concurrent Jobs in the Physical and Virtual Testbeds

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 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

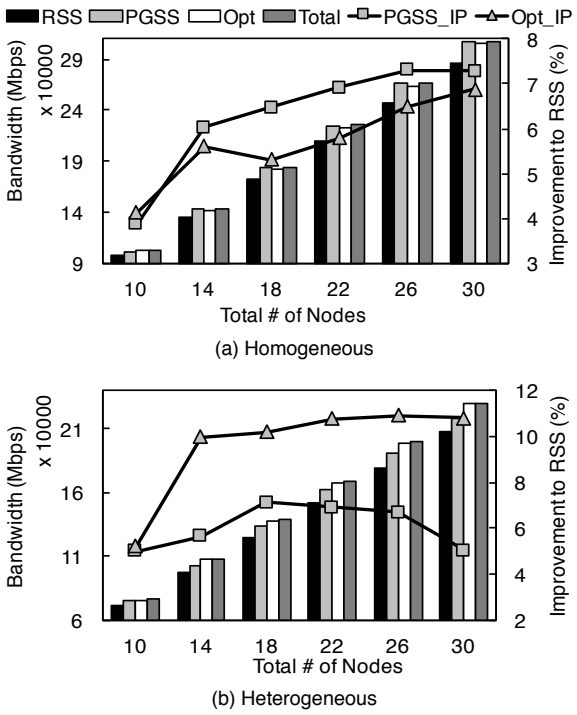


FIGURE 4.13: Bandwidth Utilization and Improvement (Suffixed with IP) of RSS, PGSS and the Online Optimal Algorithm (Opt) with Different Amount of Nodes in the Homogeneous Settings and the Heterogeneous Settings

better than RSS in bandwidth utilization with different numbers of nodes and different degrees of network heterogeneity.

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. 4.13 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

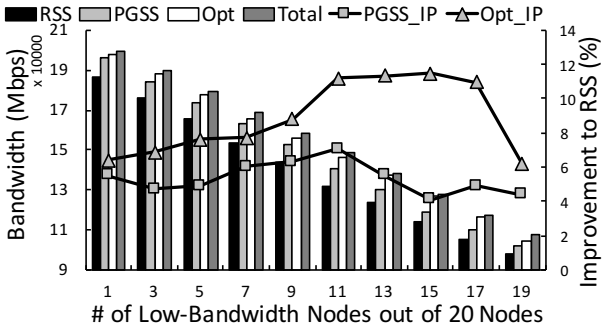


FIGURE 4.14: Bandwidth Utilization and Improvement (Suffixed with IP) of RSS, PGSS and the Online Optimal Algorithm (Opt) with Different Degrees of Network Heterogeneity

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.

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. 4.14, 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

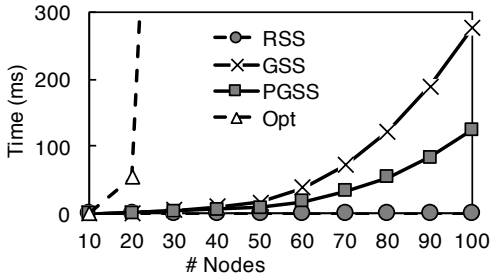


FIGURE 4.15: Scheduling Overhead Time per Flow of the RSS, GSS, PGSS and Online Optimal (Opt) Algorithms with Different Number of Nodes in the Cluster

that the application-level bandwidth-aware source selecting algorithm performs better in the heterogeneous networks.

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. 4.15 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.

4.5 Conclusion

In this chapter, 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 implement 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.

Chapter 5

Confluence: Decreasing the Data Transfer Size of Iterative Distributed Operations by Key-Dependency-Aware Partitioning

5.1 Introduction

Distributed applications consisting of iterative distributed operations are pervasive in the fields of graph computing [58, 59], database query processing [76, 85] and machine learning [49, 57]. To process the big data, distributed computing frameworks like YARN [77], Mesos [40] and Dryad [42] are often used. Several distributed computing paradigms have been developed on top of these frameworks to match various styles and scenarios of computing on large-scale data [76, 85]. MapReduce [29], being one of the most popular distributed paradigms, provides the users with a simple map-and-reduce interface that can suit most of these data analysis tasks. Generally, it saves the output data in the file system, such

as HDFS [75]. This is inefficient for tasks of the iterative distributed operations, where one iteration reuses the data from the previous iterations. The sharing of data between iterations is usually done through a shuffle operation.

Most of the distributed computing paradigms include the shuffle operation, which transfers the intermediate output data of the map operations to the computer nodes doing the reduce operations. The shuffle operation transfers data from a group of nodes to the other nodes in the same group, which follows a many-to-many communication pattern. The shuffle operation may require a large amount of network bandwidth and sometimes even dominate the overall completion time especially for the shuffle-heavy jobs, where the volume of data to shuffle is large comparing to the input data size. A study based on the Yahoo! work trace has revealed as much as 70% of the jobs are shuffle-heavy [21] and the shuffle completion time can account for as much as 33% of job completion time [9, 25]. The problem of significant shuffle workloads impedes the iterative distributed operations, where the shuffle operations transfer large volumes of data between every two iterations.

Several iterative distributed computing paradigms [18, 87] have been developed to overcome the disk I/O overheads of the computing tasks that include multiple iterations of distributed operations and thus multiple shuffle operations. Spark [87], for instance, adopts the in-memory approach to reuse data across iterations in the memory to avoid the overhead of transferring the intermediate data into and out of the disk. However, none of them proposes a useful method to minimize the size of the data to be transferred in multiple shuffle iterations.

Our key insight is that we can exploit the dependency of keys across iterations and partition the data according to the key dependency to greatly decrease the transferred data across shuffle iterations. The dataset entries are generally in the format of key-value pairs. How the output data of the previous shuffle iterations are partitioned across different locations can greatly affect the data are to be shuffled and its efficiency in the following computing iterations. By the default hashed-by-key partitioning scheme, the shuffle size of each map-and-reduce iteration would be almost of the same size as the map output data. The total shuffle

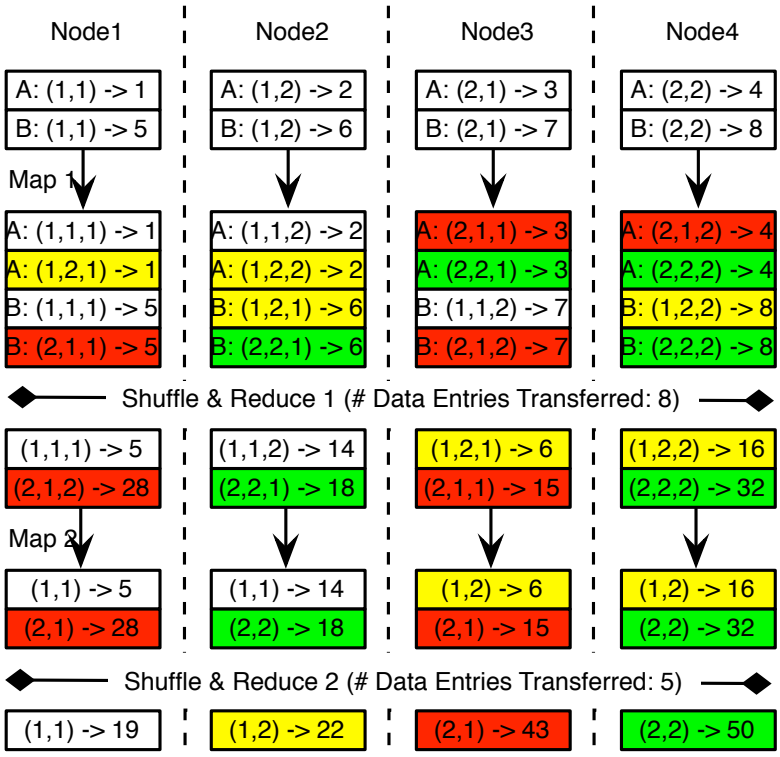
size of multiple iterations can be very large. In fact, following the semantics of the application, the key values of the data entries of the next computing iteration can usually be determined by the key values of the previous iteration. The **application semantics** refers to what new key-value data entries a key-value data entry will generate in a distributed operation, based on the logic of the program. If we have a key partitioning scheme such that some data needed for the following computing iterations can be assigned to the same computing node after shuffling, we can reduce the data volume of the shuffle operations (we call that the "shuffle size" in the rest of this chapter). We use a simple example to illustrate this idea.

For instance, the MapReduce-style matrix multiplication algorithm is a two-iteration (stage) operation whose shuffle size can be minimized with a "better" key partitioning scheme. Each entry of the matrix product $C = AB$, where $A \in R^{m \times k}$ and $B \in R^{k \times n}$, is denoted as $C_{ij} = \sum_{p=1}^{k} A_{ip}B_{pj}$. Stage 1 calculates $A_{ip}B_{pj}$, $p = 1, 2, ..., k$. Stage 2 obtains C_{ij} by summing $A_{ip}B_{pj}$, $p = 1, 2, ..., k$. Fig. 5.1 shows an example of 2×2 matrix multiplication in a four-node cluster. The data entries are in the key-value format, where the keys are (i, j, p) or (i, j). The boxes having the same color represent the related data entries needed to generate a particular result entry C_{ij}. By using the default hashed key partitioning scheme in both shuffle iterations (Fig. 5.1(b)), the data entries representing the addends for a particular sum need to be transferred to the same node in the second shuffle iteration. For example, to get $(1, 1) \rightarrow 19$, the entry $(1, 1) \rightarrow 14$ in Node 2 needs to be transferred to Node 1 to join the entry $(1, 1) \rightarrow 5$. However, if in the first shuffle iteration, knowing that entries with keys (i, j, p) are expected to all mapped to the key (i, j) in the second iteration (Fig. 5.1(c)), a better partitioning can be done such that no entry is needed to be transferred in the second shuffle iteration. For example, by assigning $(1, 1, 1) \rightarrow 5$ and $(1, 1, 2) \rightarrow 14$ to the same node (Node 1) in the first shuffle iteration, the result entry $(1, 1) \rightarrow 19$ can be obtained locally in Node 1 from the input entries in the second iteration with no data transfer.

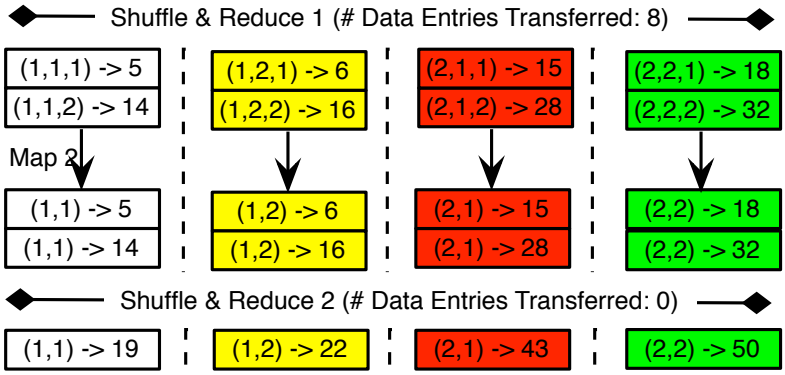
In the above simple example, obviously, there is a dependency between the keys

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

(a) Matrix A × B



(b) Hashed (Random) Key Partitioning Scheme



(c) A "Better" Key Partitioning Scheme

FIGURE 5.1: An Example of the MapReduce-style Matrix Multiplication Algorithm with Different Key Partitioning Schemes

(i, j, p) and the key (i, j). Knowing and leverage this dependency, partitioning the data entries to decrease/minimize the shuffle would become possible.

We present a new data structure the *key dependency graph* (KDG) for this purpose. The KDG is a directed acyclic graph which depicts how the keys of the dataset entries of each iteration are generated from the previous iteration. From the KDG, we can identify the subgraphs in which the dataset entries with the keys in the source nodes will only generate the dataset entries with keys in the downstream nodes. We call such a subgraph the *pure confluence subgraph*. Within a pure confluence subgraph, if all the dataset entries matching the source keys are assigned to the same node, all the generated datasets can be computed locally in the node without shuffling.

To make use of the confluence key partitioning scheme (CKP), the programs need to add the corresponding user-defined partitioner to guide the assignment of data entries to their desired locations. Most distributed computing paradigms, such as Spark [87] and Twister [33], provide an interface for adding a user-defined partitioner for the shuffle operation. By applying the CKP scheme, the shuffle sizes of multiple computing iterations can be reduced significantly. Note that this will not impact the computing workload skew level; that is, the standard deviation of the computing workloads of the nodes in the cluster after applying CKP will not be larger than that with the random hashing scheme. We are not aware of any previous work that also tried to decrease the overall shuffle traffic by a key partitioning scheme across multiple iterations based on the application semantics of the keys.

The major contributions of this chapter are listed as follows.

- We invented the new data structure Key Dependency Graph to depict the iterative application semantics and used the Confluence Key Partitioning (CKP) scheme to decrease the shuffle traffic across distributed computing iterations.
- We gave the method to analyze the data transfer size and the workload skewness when using CKP, which offers the solid shuffle traffic improvement with guarantees.

- We implemented CKP in Spark and illustrated the use of CKP with several typical iterative distributed applications from different fields. The experiment on the medium-size Spark cluster demonstrates that CKP could decrease the overall shuffle traffic workload by as much as 50%.

The rest of the chapter is organized as follows. Section 5.2 gives some background information on iterative distributed operations , dependency graph, and the model of the iterative shuffle size. In Section 5.3, we show how to construct the KDG and to reduce effectively the shuffle sizes by the Confluence Key Partitioning scheme, as well as analyzing the workload skew level. Section 5.4 introduces how to apply CKP in the real-life iterative distributed applications and Section 5.5 discusses the implementation details of CKP in Hadoop and Spark. The evaluation results of the performance are presented in Section 5.6. We conclude the chapter and suggest some possible future work in Section 5.7.

5.2 Background

5.2.1 Iterative Distributed Operations

Datasets of large volume cannot be stored in a single computer node and are often separated into several partitions, where each partition is distributed to a node in the computer cluster. We refer to a dataset which is separately stored in several nodes as a distributed dataset. The function whose inputs include the distributed dataset is called a *distributed operation*. Each entry of the distributed dataset can be represented as a key-value pair.

In many distributed computing paradigms, there are often several iterations involving different distributed operations being applied to the input dataset before the final results are obtained. Each iteration usually takes the outputs of the previous iteration as the input dataset.

One typical example of the distributed operation is MapReduce [29], which divides the operation into two primitives: map and reduce. Many iterative distributed paradigms such as the in-memory paradigm Spark [87] and the distributed database query engine Hive [76] can be equated to multiple iterations of MapReduce operations. For iterative distributed operations, we refer to the general iterative map-and-reduce operation, where the "map" primitive performs one or more transformation operations on the datasets in each local node, and the "reduce" primitive would re-partition the datasets by transferring the data entries with the same keys to the designated locations and group the entries of the same key to one key-value pair, where this *value* is a list of the values that associate with the key. The output of each data entry of the reduce primitive is also called as the key-value-list pair in this chapter. The reduce primitive does not involve any transformation on the values of the entries and is also known as the shuffle operation. Note that division of map and reduce in the matrix multiplication example in Section 5.1 is slightly different from the definition here. In the example, the reduce primitive aggregate the value list of each key to a value, which follows the traditional style of MapReduce. While by the definition here, the aggregate operations join the other data transformation operation in the map primitive of the next iteration. Each iteration of distributed operation can have a series of transformations plus at most one shuffle operation. In the rest of this chapter, the terms of the reduce primitive and the shuffle operation may be used interchangeably.

The traffic pattern of the data to shuffle in each iteration is directly dictated by the key partitioning scheme. A random or careless key partitioning scheme may cause unnecessary shuffle traffic across multiple iterations of distributed operation. This is what we set out to reduce using our partitioning scheme.

5.2.2 Dependency Graph

The dependency graph is a directed graph usually used for describing the dependency relationship of instructions, tasks, data, etc [42, 68, 90]. The scheduler can use the dependency graph to decide the order the execution of instructions or

tasks. In distributed frameworks, the scheduler uses the data dependency graph or task dependency graph to make the tasks execute in parallel in the cluster. As the tasks are meant to process their specific data, the task dependency graph is the data dependency graph as well.

In the data dependency graph, the datasets are divided into graph nodes based on locations and the dependency relationship is constructed upon the execution stages and data locations. The data partition information is specific to the application running situation and the cluster environment. In the key dependency graph proposed in this chapter, the construction of the graph does not use any data partitioning information, but considers the application semantics of the keys of the data, which depends on the application logic only.

5.2.3 Iterative Shuffle Size

We model the overall shuffle size of the iterative distributed operations and discuss the time complexity of obtaining the optimal key partitioning scheme that minimizes the overall shuffle size.

For the iterative distributed operations, the result of the i_{th} iteration can be obtained recursively by:

$$A'_i = m_i(A_{i-1}) \tag{5.1}$$

$$A_i = r_i(A'_i) \tag{5.2}$$

, where A_{i-1} is the output of the $(i-1)_{th}$ iteration as well as the input of the i_{th} iteration, m_i is the map primitive of the i_{th} iteration that operates on A_{i-1} locally without changing the partition locality, and r_i is the shuffle operation of the i_{th} iteration that takes the output A'_i of the map primitive as the input. The keys of A_i and A'_i are the same, but the partition localities are different. In the first iteration, when i is equal to 1, A_0 represents the raw input datasets before any processing.

Suppose in the i_{th} iteration of the shuffle operation, in order to collect an entry $a \in A_i$, the size of data that need to be transferred is a function of a and A'_i:

$s_i(a, A'_i)$. The function s_i can be different in different key partitioning schemes. The data transfer size to obtain A_i is:

$$S_i = \sum_{a \in A_i} s_i(a, A'_i). \tag{5.3}$$

Within the same i_{th} iteration, for $\forall a_p, a_q \in A_i$ where $a_p \neq a_q$, the values of $s_i(a_p, A'_i)$ and $s_i(a_q, A'_i)$ are independent of each other. It means that if the value of $s_i(a_p, A'_i)$ changes due to a different partition location of a_p, the value of $s_i(a_q, A'_i)$ remains unchanged unless a_q is partitioned to a different location.

If m iterations are required to compute the final result, the overall accumulated data transfer size for obtaining the final result is

$$S = \sum_{i=1}^{m} S_i = \sum_{i=1}^{m} \sum_{a_i \in A_i} s_i(a, A'_i). \tag{5.4}$$

If values of $S_i (1 \leq i \leq m)$ are independent of each other, S is a linear function and S can be minimized simply by minimizing each $s_i(a, A'_i)$. However, most usually, S_i and S_{i+1} are correlated. The value of S_i depends on the logic of the function m_i in Formula 5.1, the partition locality of dataset A_{i-1}, as well as the locations where each entry of A_i is partitioned to. The partition locality of A_i inherited from the i_{th} iteration will again affect the value of S_{i+1}.

If the contents of $A_i (i = 1, 2, ..., m)$ are already known, to find out the optimal key partitioning scheme that minimizes the overall data transfer size S, we need to exhaustively explore the possible key partition schemes of all the key-value pairs across all the iterations. By calculating the overall data transfer size of each scheme, the optimal solution is the scheme with the minimal size. The time complexity of the exhaustive method is $O(n^{\sum_{i=1}^{m} |A_i|})$, where n is the number of nodes in the cluster and $|A_i|$ is the number of data entries of A_i, which indicates that it is an NP-complete problem.

In fact, we usually do not know the contents of A_i and cannot explore the partition scheme for the next iteration until the program has actually finished the

i_{th} iteration. Due to this limitation, the idea of finding out the optimal partition schemes for all the iterations to minimize the overall data transfer size S is infeasible. In the rest of this chapter, the shuffle size is used to refer to the data transfer size of the shuffle operation.

5.3 Confluence

Even though it is hard to find the optimal partition solution to minimize the overall shuffle size, if the application semantics of the specific distributed operations (distributed operation semantics) is already known, the shuffle size can be reduced to the maximum extent by exploring the key dependency of different iterations.

This section firstly describes the the construction process of the key dependency graphs, as well as its properties. After that, we present Confluence Key Partitioning Scheme, which decreases the overall iterative shuffle size by applying the properties of the key dependency graph, while not increasing the workload skew level.

5.3.1 Key Dependency Graph

The directed graph $G = (V, E)$, which is named as the *key dependency graph* (KDG), addresses the key dependency status of the distributed operation iterations.

The *key dependency* is denoted with two vertexes $v_{i,k1} \Rightarrow v_{i+1,k2}$ if the output key-value pairs with key $k1$ in Iteration i can generate a key-value pair with key $k2$ in Iteration $i+1$. The key dependency is specific to the distributed operation semantics and the distributed operation semantics decides which graph node should point to which others. The distributed operation semantics should be known prior by the programmers.

Algorithm 2: Construction of the Key Dependency Graph

Input : Key Set K_i, containing all the possible keys (or key patterns) of the output key-value pairs of any Iteration i;
Key Dependency Set D, containing the key dependency of all iterations;

Output: Key Dependency Graph $G = (V, E)$

```
/* Construction of V                                          */
1  foreach Iterations i do
2  |  foreach k ∈ K_i do
3  |  |  add vertex v_{i,k} to V
4  |  end
5  |  if any possible key is unknown then
6  |  |  add vertex v_{i,x} to V
7  |  end
8  end
/* Construction of E                                          */
9  foreach Iterations i do
10 |  foreach key dependency v_{i,k1} ⇒ v_{i+1,k2} in D do
11 |  |  add edge (v_{i,k1}, v_{i+1,k2}) to E
12 |  end
13 end
```

FIGURE 5.2: Pseudo Code of Constructing the Key Dependency Graph

The construction process of the key dependency graph G is shown as in Algorithm 2. Note that when constructing E, for the unknown keys, it is assumed that $v_{i,x} \Rightarrow v_{i+1,k2}, \forall k2$.

This graph is a directed acyclic graph (DAG). The time complexity to construct this graph is $O(\sum_{i=1}^{m} k_{i-1}k_i)$, where k_i is the number of keys (or key patterns) in iteration i and m is the total number of iterations. Usually, k_i is very small in value when it is referred as the key pattern (not the distinguished values of the keys), which is constrained by the application semantics. For example, in the matrix multiplication application, the subgraph from the second level to the third (final level) of the KDG can be denoted as a one node to one node subgraph: $(a, b, [1, n]) \rightarrow (a, b)$, where a and b stand for any row and any column index of the matrix, and $[1, n]$ stands for the range from 1 to the the matrix dimension length n.

The subgraph $C = (V', E')$ of $G = (V, E)$ is called a *confluence subgraph* if it satisfies the following *outward exclusive* requirements: denote V'_i as the set of all the vertexes in V' which are in level i in G. V'_i satisfies $V'_i = \bigcup_{v'_{i-1} \in V'_{i-1}} V_i$, where V_i is the set of vertexes that v'_{i-1} has an edge $e \in E$ points to. The edge $e' = (v'_1, v'_2)$ is in E' iff $v'_1 \in V', v'_2 \in V'$ and $(v'_1, v'_2) \in E$.

In other words, in the confluence subgraph, all the vertexes in the upper level have no edge pointing to the vertexes in the lower level outside the confluence subgraph. Let Level u and Level w be the uppermost level and the lowest level that V' contains, respectively. V'_u is called the *source key set* at level u and V'_w the *confluence key set* at level w of the confluence subgraph $C = (V', E')$.

We call the confluence subgraph $C = (V', E')$ the *pure confluence subgraph* if it further satisfies the *inward exclusive* requirement: $V'_{i-1} = \bigcup_{v'_i \in V'_i} V''_{i-1}$, where V''_{i-1} is the set of vertexes that have an edge $e \in E$ pointing to v'_i.

An example of the KDG of 3 levels and some of its confluence subgraphs are shown in Fig. 5.3. The subgraph in the dashed boxes C3 is a pure confluence subgraph of C1. The subgraph in C2 is a confluence subgraph but not a pure confluence subgraph, as the source node of the edges $(v0x, v11)$ and $(v0x, v12)$ is not in C2. The subgraph in the dotted box C4 is not a confluence subgraph as the edge $(v0x, v12)$ points to the node $v12$, which is not a vertex in the subgraph C4. Note that the key dependency graph is always the pure confluence subgraph of itself, as show in the dashed box C1.

5.3.2 Properties of Confluence Subgraph

For the distributed operations of several iterations, suppose that there is a confluence subgraph C of its KDG with the source key set V'_u at level u and the confluence key set V'_w at level $w (w > u)$. Let B_u be a part of the output dataset of the u_{th} iteration and B_w be the dataset generated by B_u after $(w - u)$ iterations. We have a theorem that can easily be conducted by the definition of the confluence subgraph.

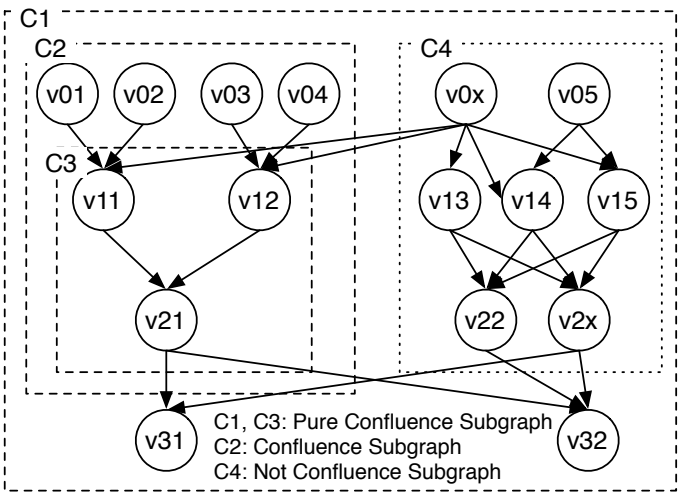


FIGURE 5.3: A Key Dependency Graph Example

Theorem 5.1. *If the keys of all the entries of B_u are in the source key set V'_u, the keys of all entries of B_w are in the confluence key set V'_w.*

We call the datasets B_u and B_w in Theorem 5.1 the *upward closure pair* datasets of the confluence subgraph. It can be further conducted from Theorem 5.1 to get:

Theorem 5.1.1. If the subgraph C is a pure confluence subgraph and the keys of all the entries of B_u are in the source key set V'_u, when B_u is stored in one single node, the minimum data transfer size for obtaining B_w is 0.

Theorem 5.1.1 indicates that if the datasets B_u and B_w are the upward closure pair of a pure confluence subgraph and B_u is partitioned in one node, we can finish all the following $(w - u)$ iterations of distributed operations in the same node to obtain A_w, without any data transferring. We name this key partitioning scheme the Confluence Key Partitioning scheme.

5.3.3 Confluence Key Partitioning

We demonstrate that the overall data transfer size can be decreased by localizing the pure confluence subgraph, i.e., placing the datasets corresponding to the pure

confluence subgraph in a node.

Assume that there is a pure confluence subgraph C in the KDG, from level u to level $w(u \neq 0, w > u)$. Let V'_u and V'_w be the source key set and the confluence key set of C, respectively, and B_u and B_w be the upward closure pair datasets. The *Confluence Key Partitioning (CKP)* scheme partitions the dataset B_u in one single node in Iteration u and completes the following $(w - u)$ iterations of operations on B_u and its generated dataset in the same node.

CKP localizes the data in the granularity of a pure confluence subgraph, different pure confluence subgraphs are partitioned randomly.

Although the KDG is the pure confluence subgraph of itself, we do not consider the localizing the whole KDG in the above discussion. The reason is that the raw input dataset is always distributed across the cluster before the first iteration.

We analyze the expected overall data transfer size S' of CKP in comparison with that S'' of the random key partitioning scheme (RKP), which is ignorant of the key semantics and partition the data entries randomly (e.g., partition by the hashed value of keys). S'_i and S''_i denote the expected data transfer size of CKP and and RKP in the i_{th} iteration, respectively.

Before going further, we figure out the function $s_i(a, A'_i)$ in Formula 5.3 in the case that a is randomly (uniformly) partitioned to a node. In this case, the probability of a is partitioned to any node is $1/n$, where n is the number of nodes in the cluster. We call such an entry that has the same probability being partitioned to any node as the uniformly-partitioned entry. As long as the partition location of an entry is finally hashed to the computer nodes, the entry is a uniformly-partitioned entry.

Suppose that a' is the set of input data entries that have the same key as a right before the shuffle operation and p_j denotes the percentage of data entries of a' that lie in Node j, where $\sum_{j=1}^{n} p_i = 1$. We have

$$s_i(a, A'_i) = \sum_{j=1}^{n} [\frac{1}{n}|a|(1 - p_j)] = \frac{(n-1)}{n} \cdot |a| \tag{5.5}$$

, where $|a|$ is the length of the value list of a, which is also the number of data entries in a'. It shows that if a is randomly partitioned to a node, the distribution of a' does not affect the data transfer size for collecting a.

Theorem 5.2. *The data transfer size for collecting the uniformly-partitioned entry in the shuffle operation is decided only by the number of nodes in the cluster and the length of this entry.*

By Theorem 5.2, when using RKP, the data transfer size of the i_{th} iteration is $S''_i = \sum_{a \in A_i} \frac{(n-1)\cdot|a|}{n} = \frac{n-1}{n}|A'_i|$, where $|A'_i|$ is the number of data entries in A'_i.

For CKP, we analyze every iteration as follows.

1) From Iteration 1 to $(u-1)$, the key partitioning scheme is the same as RKP: $S'_i = S''_i, 1 \leq i \leq u-1$.
2) In Iteration u, an entry of B_u belongs to a pure confluence subgraph and the pure confluence subgraph is randomly partitioned. The entries in B_u are uniformly-partitioned entries, as well as the entries in $A_u - B_u$. Therefore, $S'_u = S''_u$.
3) From the $(u+1)_{th}$ to w_{th} iteration, the data transfer size of the operations on the datasets generated by B_u is 0, by Theorem 5.1.1. We have $S'_i = S''_i - \frac{n-1}{n}|B'_i|$, where $r_i(B'_i) = B_i, u+1 \leq i \leq w$.
4) From iteration $(w+1)$ onward, the data entries are uniformly-partitioned entries again: $S'_i = S''_i, i \geq w+1$.

In all, the expected overall data transfer size after applying CKP is $S' = S'' - \frac{n-1}{n}\sum_{i=u+1}^{w}|B'_i|$, which eliminates exactly the shuffle workload of the datasets that associate with the keys in the pure confluence subgraphs as compared to RKP, without introducing other shuffle workloads. $\sum_{i=u+1}^{w}|B'_i|$ stands for the shuffle size improvement of CKP as compared to RKP. This result indicates that CKP can reduce more data transfer size if the pure confluence subgraph contains more iterations or the size of the dataset corresponding to the pure confluence subgraph is larger. In Section 5.4, the value of $\sum_{i=u+1}^{w}|B'_i|$ will be figured out for the specific distributed operations.

5.3.4 Workload Skew Analysis

The workload of a node is the number of data entries of the map primitive in that node and The workload skew level of Iteration i is the standard deviation of the expected workloads of different nodes in that iteration, which is denoted as L_i. The math symbols in Section 5.3.3 are reused here without duplicated definition. Only the data entries that associate with the pure confluence subgraphs are counted as the other data (if any) are randomly partitioned in both RKP and CKP and will not affect the standard deviation of the workloads. And we only consider the workload skew levels of the iterations that the pure confluence subgraphs affects (Iteration $u+1$ to $w+1$). The reason is that both RKP and CKP use the same key partitioning policy in the CKP-not-affected iterations and the workload skew levels of these iterations will be the same in both schemes.

In all iterations when RKP is applied, every entry is a uniformly-partitioned entry, the expected workloads of the all the nodes are the same and the expected workload skew level is 0.

When CKP is applied, as each output entry of the shuffle operation is the uniformly-partitioned entry in Iteration u, the workloads of all the nodes are the same in Iteration $u+1$. Therefore, $L_{u+1} = 0$.

In Iterations from $u+2$ to $w+1$, it can be divided into two phases based on the semantics of the distributed operations. Suppose that Iteration $u'(u+1 \leq u' \leq w)$ is such an iteration that in every iteration from $u+1$ to $u'-1$, each input entry of the map primitive is expected to generate the same number of output entries of the shuffle operation, and in Iteration u', the expected number of output entries of the shuffle operation generated by each map input entry is unknown or not the same. Note that such Iteration u' may not exist and in this case, we denote it as $u' = w+1$.

Phase 1: From Iteration $u+2$ to u', the workloads are the same in all the nodes and the workload skew level of every iteration is 0.

Phase 2: From Iteration $u'+1$ to $w+1$, the workloads of the nodes in each iteration are unpredictable, which are dependent on the number of data entries

that can be generated by the data localized in each node. Therefore, $L_i \geq 0, u' + 1 \leq i \leq w + 1$.

Although the workload skew levels of CKP in Phase 2 is unpredictable, fortunately, such Iteration u' does not exist in many distributed operation, including matrix multiplication and the other distributed applications (MovieLensALS, MultiAdjacentList and KMeans) introduced in Section 5.4. In these applications, the expected workload skew level of CkP is 0, which is the same as that of RKP. In the distributed application where u' does exist, the workload skew level may not be 0 and specific workload skew analysis is needed for this specific application.

5.4 Application

In this section, we show how CKP can be applied in various iterative distributed applications, even when the applications are not well-structured and the pure confluence subgraphs are not obvious, by dividing the operations into multiple sub-operations.

In the key dependency graph of the matrix multiplication algorithm, the pure confluence subgraphs are obvious in the second iteration, whose keys follow the many-to-one mapping pattern. In some cases, CKP may not be applicable when the pure confluence subgraphs cannot be foundin the straightforward way. In this case, if the iterative distributed operations can be divided into several smaller operations to fit the Confluence model, CKP can be flexibly applied to some of the divided operations. The division of the iterative distributed operations can be based on either the operations or the datasets. In the following, we illustrate the methods to divide the iterative distributed operations with the real-life application examples, namely MovieLensALS, MultiAdjacentList and KMeans.

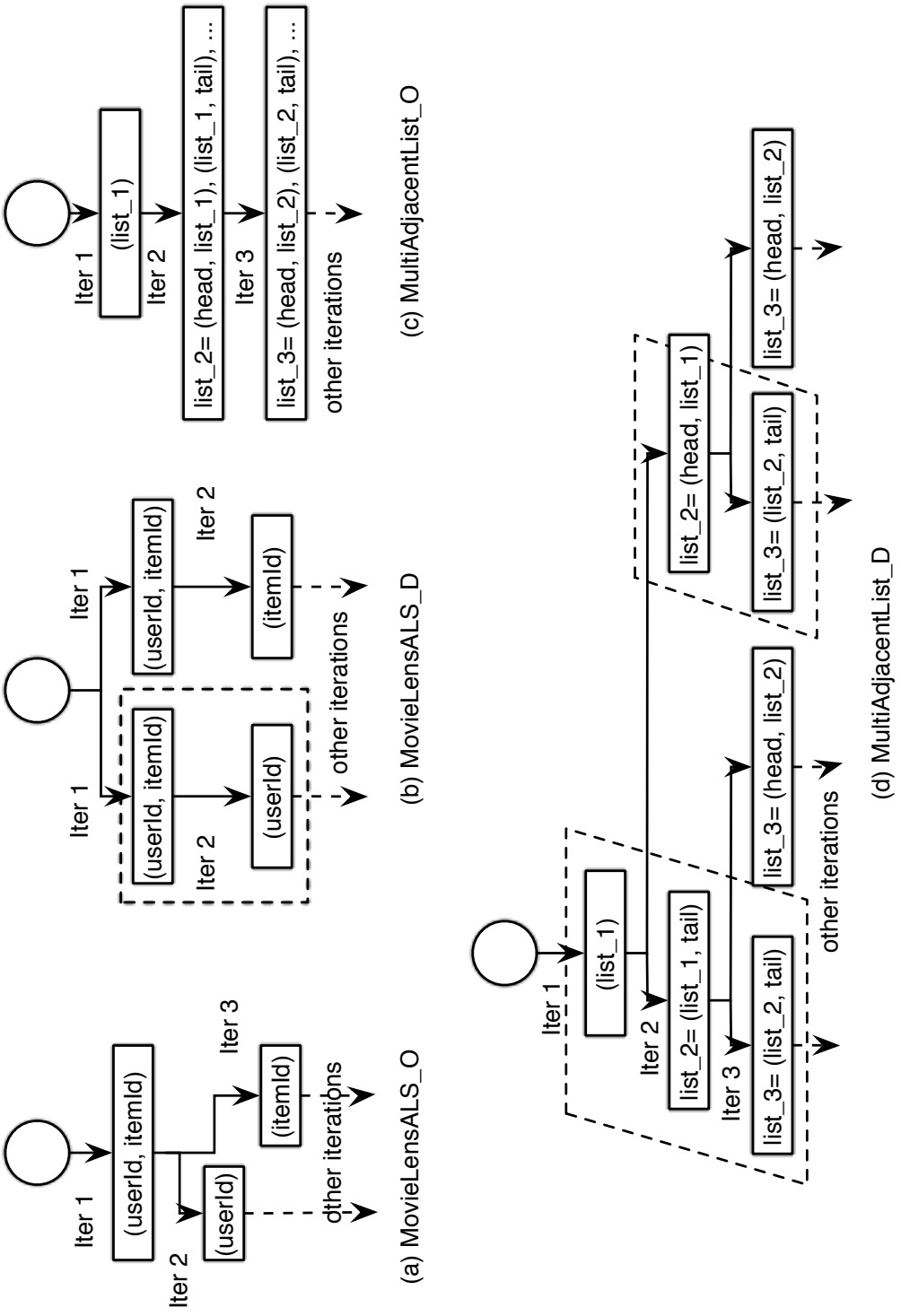


FIGURE 5.4: The Original (Suffixed by _O) and Divided (Suffixed by _D) Iterative Evolution of the Datasets of the MovieLensALS and MultiAdjacentList Applications. The Keys of the Datasets are Marked by Parenthesis. Pure Confluence Subgraphs can be Found in the Computing Iterations in the Dashed Boxes

TABLE 5.1: CKP Shuffle Size Improvements in Different Distributed Operations

Distributed Operations	**CKP Shuffle Size Improvement**
MatrixMultiplication	$n^3 for R^{n \times n}$
MovieLensALS	# of $(user, item)$ entries
MultiAdjacentList	1/2 the RKP shuffle size in every iteration
KMeans	$\|A\| \cdot \sum_{i=2}^{m}(\sum_{j=1}^{k} p'_{ij} - 1/n)$ upper bound:$\|A\| \cdot (m-1)(n-1)/n$ lower bound:$-\|A\| \cdot (m-1)/n$

5.4.1 MovieLensALS

MovieLensALS [3] is a recommendation application that uses Alternating Least Squares (ALS) method in collaborative filtering in order to recommend the product items to the users based on the user-item rating information. During the execution, it organizes the rating data into the user-item blocks grouped by the keys "$(userId, itemId)$". In the successive iterations, it needs to reorganize the user-item block data by generating the user block and the item block grouped by "$userId$" and "$itemId$", respectively. This operation is done in two iterations as shown in Fig. 5.4(a). No single key partitioning scheme can remove the shuffle data in order to generate both of these blocks from the user-item block, as the key dependency follows a many-to-many pattern. However, if the iterative distributed operation is regarded as two individual iterative operations and is divided as in Fig. 5.4(b), the pure confluence subgraph can be found in the operation in the dashed box. CKP can then be applied on the key dependency $(userId, itemId) \Rightarrow userId$ for the user block.

As CKP is applied only in one successive iteration in MovieLensALS, the workload skew level is 0.

5.4.2 MultiAdjacentList

MultiAdjacentList [4] is a graph computing application which generates the heads and tails of lists of various lengths from the information of edges. In every

iteration, where the key L_i represents a list of length i and its values are the sets of the head nodes and tail nodes of the list, it generates a group of new lists of length $i+1$ by appending each head node and each tail node to it. Each new list acts as the key and its new head nodes and tail nodes act as the value. The evolution of the data of multiple iterations is shown as Fig. 5.4(c). The key dependencies are $L_i \Rightarrow (heads, L_i)$ and $L_i \Rightarrow (L_i, tails)$. We cannot use CKP to localize the keys $(heads, L_i)$ and $(L_i, tails)$ with L_i, as the datasets corresponding to Key $(heads, L_i)$ and Key $(L'_i, tails)$ may overlap for different L_i and L'_i. However, the iterative operations can be divided into several smaller iterative operations by dividing the dataset as in Fig. 5.4(d). In the iterative operations in the dashed boxes, the pure confluence subgraphs exist in their KDG's by the key dependency $L_i \Rightarrow (L_i, tails)$. Applying CKP to these operations and leaving the other iterative operations that are related to the keys $(heads, L_i)$ to be randomly partitioned, only half of the datasets need to be shuffled.

In the setting that every node has l heads and l tails on average, every map input entry is expected to generate $2 \cdot l$ shuffle output entries in each iteration. Therefore, by the discussion in Section 5.3.4, the workload skew level of each iteration is 0 when CKP is applied.

5.4.3 KMeans

KMeans [6] is the popular clustering algorithm in data mining, which groups the vector points into k clusters where each vector point belongs to the cluster with the nearest mean. This process could be done in a dedicated number of iterations. With the initial k chosen points regarded as the centers of the clusters, in each iteration, every point is compared with the k centers and is is grouped into the cluster where the Euclidean distance between the point and the center is the smallest. After all points are grouped, each new cluster center for the next iteration is chosen by the mean of the points in that cluster. The shuffle operation takes place when calculating the mean of each cluster whose points are located in different nodes.

If the points are bound to the same cluster in every iteration, by using the key dependency $points \Rightarrow centers$ and localizing the points in the same cluster at the first iteration, CKP can eliminate the shuffle size in the successive iterations. However, the affiliation of point to clusters are not stable between iterations. One point that is grouped into one cluster in a iteration can be grouped into another cluster in the next iteration. We estimate the shuffle size in this case.

Suppose that p'_{ij} is the percentage of points that are grouped in Cluster j both in the first CKP iteration and Iteration $i(i \neq 1)$, $|A|$ is the total number of points, m is the total number of iterations, and n is number of nodes. The shuffle size of Iteration i is $S_i = |A| \cdot (1 - \sum_{j=1}^{k} p'_{ij})$. The CKP shuffle size improvement in Iteration i is $(n-1)/n \cdot |A| - S_i = |A| \cdot (\sum_{j=1}^{k} p'_{ij} - 1/n)$, and the overall CKP shuffle size improvement is $|A| \cdot \sum_{i=2}^{m} (\sum_{j=1}^{k} p'_{ij} - 1/n)$. The higher $\sum_{i=2}^{m} \sum_{j=1}^{k} p'_{ij}$ is, the higher the shuffle size improvement. In the best case that points are bound to the same cluster in every iteration, $\sum_{j=1}^{k} p'_{ij} = 1(i = 2, 3, ...m)$ and the shuffle size improvement is $|A| \cdot (m-1)(n-1)/n$. In the worst case, every point grouped into cluster i in the first iteration is grouped to a cluster other than i, the shuffle size improvement is $-|A| \cdot (m-1)/n$. The affiliation of the points to the clusters are more stable after running a few iterations. Instead of localizing the points of the same cluster in the very first iteration of KMeans, localizing them after a few beginning iterations can give higher $\sum_{j=1}^{k} p'_{ij}$.

In every iteration of KMeans, as every input point generates the same point as the input for the next iteration, the workload skew level of every iteration is 0.

A summary of the CKP shuffle size improvements of the distributed operations discussed above is shown in Table 5.1. All the values of the CKP shuffle size improvement are figured out without consideration to the map-side combine mechanism discussed in Section 5.5.

5.5 Implementation

This section introduces the effect of the map-side combine mechanism and the limitations of applying CKP in MapReduce. The implementation details of binding partitions and executors to facilitate CKP in Spark and the discussion on how to automate the confluence approach will also be presented.

5.5.1 Map-Side Combine

In practice, existing distributed frameworks can do map-side combine for the map output data before the shuffle operation if the reduce operation is an aggregate function, e.g, the sum function of the reduce operation in Stage 2 of the matrix multiplication algorithm as mentioned above. The map-side combine does the local reduce operation on map output data such that the data needed to transfer in the shuffle operation can be greatly decreased in size even when RKP is applied. However, CKP is still beneficial because it eliminates the overhead time of initializing and cleaning up the shuffle connections, which can be time-consuming.

5.5.2 Confluence in Hadoop

CKP can be widely applied to decrease the shuffle size of the iterative distributed paradigms that compute and store intermediate datasets in the memory, like Spark and Twister. However, we find it not applicable to Hadoop MapReduce for the following reason. Hadoop MapReduce stores the output result of each distributed operation iteration in the distributed file system, i.e., HDFS, where the partitioned locations of the data blocks are reorganized due to the internal storing mechanism of HDFS. Even if the datasets are partitioned by CKP during the distributed operation, the partitioned locations of the datasets will not be retained in the next iteration of the distributed operation.

An alternative solution is to modify the mechanism of HDFS on choosing the partitioned locations of data blocks when storing data to a file. If the dataset is

able to declare the preferred partitioned location as the node where it is kept in the memory and HDFS writes the dataset to its preferred partitioned locations, the partitioned locations of the datasets will remain unchanged and CKP is applicable. We have not implemented this new mechanism in HDFS and the feasibility is yet to be verified.

5.5.3 Binding Partition and Executor in Spark

In the implementation of Spark, the partition is only a logical location where the task executes the dataset, while the logical locations of tasks are not tied up with the actual(physical) positions of the executors. In other words, the tasks working on the same partition are not guaranteed to be assigned to the same executor (or the same computer node) across different iterations, in which case shuffling of data is still needed.

To amend the mismatch of the logical partition and the actual position of the executor, we implement the new feature in Spark to allow the binding of the task partitions and the executors. The binding can be done in the hash manner. In the default settings of the task scheduler of Spark, when an executor becomes free, the task scheduler selects the task with whichever partition from the head of task queue. The selected task will run in that executor. Now, if the partition-executor-binding feature is turn on, when the task scheduler selects a task for the free executor, it selects the task whose task partition hash code is the same as the hash code of the free executor. The hash manner binding ensures that the tasks of the same partition will always be allocated to the same executor in different iterations. Finding the first task that first hashed maps the executor from the task queue, the time complexity of scheduling each task is $O(N)$, where N is the number of the executors in each computing iteration.

The partition-executor-binding implementation will not introduce negative side effects into the system in the following three aspects: 1) The order to run the pending tasks of the same iteration does not affect the completion time of each computing iteration; 2) In the hash manner, each executor is expected to run

the same number of tasks in each computing iteration; 3) The task scheduling overhead is trivial and negligible, which is only linear to the number of executors.

5.5.4 Automating Confluence

Currently, programmers need to manually discover the pure confluence subgraphs of the iterative distributed applications from KDG and specify the key partitioning scheme in the program code if they apply CKP. A system which automates this process before the runtime of the application and frees the programmers the extra work may sound attractive. However, there are some difficulties implementing such a system because it is hard for the machines to really **understand** the application semantics. The reasons are as follows. 1) Difficulty in constructing the vertexes: without the provision of the real meanings of the keys by the programmer, the domain of the values of keys or the patterns of the keys are unknown without visiting the values of the whole dataset. 2) Difficulty in constructing the edges: the key dependency relationship is unclear by merely looking at logic of the program code because the key dependency relationship can also depend on the values, which are unknown until runtime.

To apply CKP, the powerful partitioner interface offered by the pervasive distributed frameworks allows the programmers to easily implement the specific partitioner for their distributed application. We make one step further by implementing the half-automatic approach so that if the programmers provide the information of the pure confluence subgraphs via a simple interface, the system will generate the corresponding partitioner and apply CKP automatically.

We implement the class named ConfluencePartitioner in Spark, which only requires the programmer to input the user-defined function that defines the mapping of the keys to the pure confluence subgraphs. The shuffle operation uses the ConfluencePartitioner to identify the pure confluence subgraphs that the data entries belong to and partitions each pure confluence subgraph randomly by the hashed value of the subgraph. To apply CKP with ConfluencePartitioner can be as simple as a single line of code in the Scala language:

TABLE 5.2: User-Defined Functions of ConfluencePartitioner in Different Distributed Operations

Distributed Operations	**Key Dependency**	**User-Defined Function**
MatrixMultiplication	$(i, j, p) \Rightarrow (i, j)$	(a:Any) => a match {case (_ , _, _) => (a._1, a._2)}
MovieLensALS	$(user, item) \Rightarrow user$	(a:Any) => a match {case (_ , _) => a._1}
MultiAdjacentList	$list \Rightarrow (list, tails)$	(a:Any) => a match {case s:String => {s.split(" ")(0) }}
KMeans	$points \Rightarrow cluster$	(a:Any) => a match {case p:Vector[Double] => closestPoint(point, cluster_centers)[1]

[1] closestPoint(point, cluster_centers) is the function which returns the index of the point in "cluster_centers" that is in the shortest Euclidean distance with "point".

data.shuffleOperation (new ConfluencePartitioner (num-Partitions, user-defined-function))

Table 5.2 shows the user-defined functions in the Scala codes of different distributed operations mentioned above.

To calculate the partition of a key-value entry in the shuffle operation, the user-defined function is applied to the key of the entry, and the returned result is then hashed by number of total partition nodes. The hashed value decides the partition location of this entry. In this way, the key-value entries whose returned values of the user-defined function are the same are considered to match the same confluence key node and will be partitioned to the same node.

With ConfluencePartitioner, the programmers can apply CKP even without knowing the details of implementing the self-defined partitioner and repeatedly writing different self-defined partitioners for different distributed operations.

TABLE 5.3: Benchmark Settings

Benchmark	**Input Data**	**Runtime Setting**
MatrixMultiplication	matrix type: $Z^{1000\times1000}$	Nil
MovieLensALS	21,622,187 ratings from 234,934 users on 29,584 movies	32 user blocks and 32 item blocks
MultiAdjacentList	25,000,000 vertexes with average in/out-degree being 2	4 iterations
KMeans	64,534,480 wikipedia page visit records	32 centers, 10 iterations

5.6 Evaluation

We conduct experiments on a physical testbed to compare the performance of CKP with the default RKP scheme in various aspects: the improvement on the shuffle sizes of different iterations, the shuffle skew level of the executors, and the scalability via the metrics of the overall shuffle sizes against different input data volumes.

5.6.1 Testbed and Benchmarks

The testbed consists of 18 computer nodes of the Gideon-II cluster in HKU [2], where each node is equipped with 2 quad-core, 32 GB DDR3 memory and 2 $\times$ 300 GB SAS hard disks running RAID-1. The nodes run with Scientific Linux 5.3 and are connected to an internal non-blocking switch with GbE ports. In the setting of the YARN cluster, one node takes the role of the name node of HDFS and one other acts as the resource manager of YARN, while the remaining 16 nodes are configured as both the HDFS data nodes and YARN node managers. Spark is deployed on top of the YARN cluster with 16 executor, where each executor runs with 8 GB of memories.

Several benchmarks that are representative ones of their fields are used to evaluate the performance of CKP. Unless further specified, the input data sizes and the running setting of the benchmarks are listed in Table 5.3. Pure confluence subgraphs can be found based on the key dependency relationship indicated in the table, and CKP is applied on these pure confluence subgraphs. CKP is compared with RKP as the baseline as RKP is the default and generally used key partitioning scheme for the distributed operations. Other key partitioning scheme are sometimes used to balance the workload of the executors for specific datasets. These key partitioning schemes rely not only on application semantics but also on the distribution of the datasets, we do not compare CKP with them here.

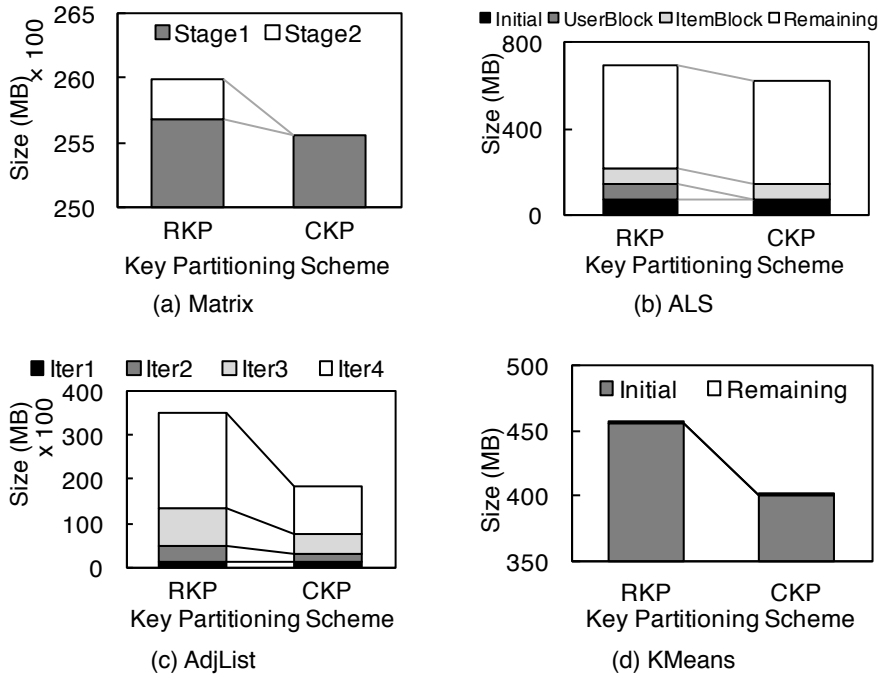


FIGURE 5.5: Shuffle Sizes of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

5.6.2 Metrics

We measure the shuffle size of each distributed computing iteration (or alternately, stage) of benchmarks. The shuffle size of each iteration is the sum of the data transfer size of the shuffle operation of all the executors in that iteration. This metric depicts the performance of CKP in decreasing the data transfer size. Besides, the standard deviation of the shuffle sizes of the executors is measured as the metric to evaluate the workload skew level of the key partitioning schemes. This metric is also measured in each iteration of the benchmark. The scalability of CKP is measured by the overall shuffle size of all the iterations that are in the pure confluence subgraphs with different volumes of input data.

5.6.3 Shuffle Sizes Improvement

The results of the shuffle size of each iteration (or stage) after applying CKP and RKP in all the benchmarks are shown in Fig. 5.5, respectively.

In MatrixMultiplication, CKP totally removes the shuffle operation of Stage 2 whose shuffle size is 0, and the shuffle size of Stage 1 remains closed to that of RKP (Fig. 5.5(a)). Note that when RKP is applied, the shuffle size of Stage 2 is very small as compared to Stage 1 due to the map-side combine mechanism as mentioned above.

The benefit of applying CKP on the smaller iterative distributed operation divided based on operations can be read in Fig. 5.5(b). In MovieLensALS, after applying CKP, the shuffle size for generating the user block is 0, while that for generating the item block remains almost the same as that of RKP. As a result, the CKP shuffle size for generating these two block is half of that of RKP. Because the other iterations of the distributed operations cannot be improved by CKP, the overall shuffle traffic CKP can decrease for the application is about 11%.

After applying CKP in MultiAdjacentList benchmark, the shuffle size of each iteration is decreased by half beginning from Iteration 2 (Fig. 5.5(c)). The reason is that by using CKP on divided sub-operations that handle the datasets corresponding to the key $(list, tails)$, only the datasets of Key $(heads, L_i)$ need to be shuffled.

In the KMeans benchmark, by partitioning the points into the clusters they belong to at the first few iterations, CKP decreases the overall cluster shuffle size by 12%. This decrease in shuffle size is contributed by avoiding the repartition of the points that are stable to their clusters, although not all points are fixed to their clusters in every iteration.

Note that the shuffle sizes of the other iterations that do not apply CKP remains almost unchanged, which means that CKP can decrease the shuffle size of distributed iterations that have the confluence key dependency without introducing extra workloads to the other iterations. How CKP can decrease the

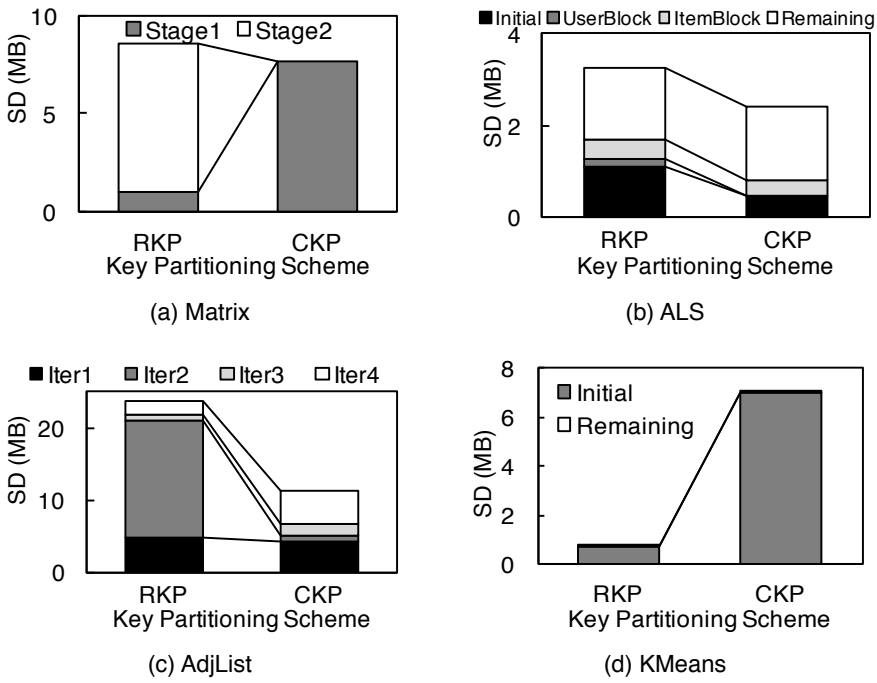


FIGURE 5.6: Standard Deviations (SD) of the Shuffle Sizes of the Executors of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

overall shuffle traffic depends on how large the portion of data corresponding to the pure confluence subgraph take in the overall processing of the application. For MultiAdjacentList, the CKP can be applied to half of the operations, either appending to the head or to the tail. While for MovieLensALS, CKP can only be applied to the operation that arrange the user-item blocks, which is only a small portion of the overall application. Still, it indicates the potential of applying CKP in the complicated iterative distributed applications.

5.6.4 Workload Skew

The standard deviations of the shuffle workloads of the executors of different benchmarks are shown in Fig. 5.6. As expected, the workload standard deviations of the CKP-affected stages of MatrixMultiplication (Fig. 5.6(a)), MovieLensALS (Fig. 5.6(b)) and MultiAdjacentList (Fig. 5.6(c)) are all 0. In the KMeans benchmark, the standard deviation of CKP is larger than that of RKP (Fig. 5.6(d)). The reason is that the number of points belonging to each cluster varies but the number of cluster centers (32) is not large enough as compared to the computer nodes (16) to even the workloads of the nodes. If the number of cluster centers is large enough, the actual number of points partitioned to each node will get closer to the expected value, which is the same in every node. Still, the standard deviation of CKP (7 MB) is small as compared to the mean shuffle size of the executors (25 MB).

For the stages that are either before or after the CKP-affected stages, the standard deviations remain almost the same as RKP, or the change is so tiny as compared to the overall shuffle size of the stage that can be ignored.

5.6.5 Scalability

The overall shuffle sizes of the pure-confluence-subgraph-related iterations with different input sizes in various benchmarks are shown in Fig. 5.7. In MatrixMultiplication, the overall shuffle size of RKP increases linearly as the matrix length grows, while that of CKP is always 0. CKP can totally remove the shuffle workload for Stage 2 of MatrixMultiplication. In the other benchmarks, the shuffle sizes of CKP always keep in a fixed (or stable) ratio to those of RKP, e.g., 50% in both MovieLensALS and MultiAdjacentList and around 88% in KMeans. The result shows that CKP scales well with different volumes of input data. The CKP decreases the shuffle workload greatly (about 50%) in one iteration out of the many distributed operations in the MovieLensALS application, and a little (about 12%) but in every iteration of the distributed operations of the KMeans application.

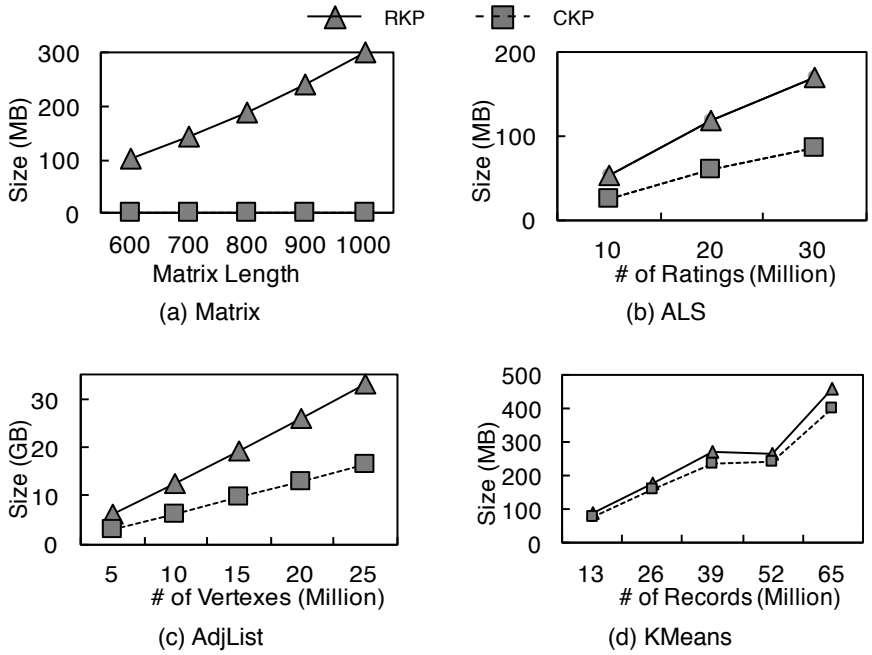


FIGURE 5.7: Overall Shuffle Size of the Pure-Confluence-Subgraphs-Related Iterations in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks with Different Input Size

5.7 Conclusion

We have presented Confluence Key Partitioning scheme, the first work on decreasing the shuffle size of iterative distributed operations leveraging the key dependency. Confluence guides the data partitioning across different iterations based on the key dependency graph, which is constructed with the application semantics. Confluence greatly decreases the overall shuffle size while not introducing the workload skew side effect for a variety of distributed operations in the fields of scientific computing, machine learning, data analysis, etc.

Chapter 6

Conclusion and Discussion

6.1 Conclusion

This thesis discusses the problems of distributed computing platforms thoroughly from the framework to the application and provides the solutions to improve the performance of various distributed computing scenarios.

First, we explore in depth the reasons of the underutilization of the cluster computational resources and network resources, and propose SMapReduce and BAShuffler to make full utilization of these two types of resources, respectively. By the fully understanding of the thrashing phenomenon and the performance factors to the MapReduce applications, SMapReduce manages the working slots dynamically based on the runtime performance feedbacks so as to produce the maximum computational throughputs of the jobs. BAShuffler, on the other hand, maximized the network bandwidth utilization by scheduling the MapReduce shuffle source nodes based on the max-min fairness bandwidth estimation. SMapReduce and BAShuffler are optimized for the computation-heavy jobs and communication-heavy jobs respectively.

Further, we bridge the gap between the distributed application semantics and the cluster task assignment policies in order to decrease the cluster network traffic workloads. By exploring the data key dependency in different stages in

the iterative distributed computing applications, we propose Confluence Key Partitioning scheme to partition data depending on the key dependency graph and the traffic workloads via network can be greatly alleviated.

The three techniques enhance the distributed computing in different aspects, independently. They combined provide the integrated solution to high-performance distributed computing from framework to application.

6.2 Discussion

Before ending this thesis, the author would like to discuss some thoughts on the work of improving the distributed systems and some future trends.

About the research problems: Compared to other systems, the complexity of the distributed systems is almost always beyond imagined and the application use cases usually vary greatly because of different use purposes, it is always not an easy job to shoot the bottleneck of the distributed systems. A full understanding of the distributed system from architecture, rational and mechanism to implementation logic and application may be the precondition for finding the potential bottleneck. Experiments with real application can work to verify the conjecture, , help in discovering the hidden problems, as well as to inspire people the possible solution to overcome the problem. A broad survey of the other similar systems can reveal how common the problem is in the ecosystem, and how many applications used by the community and industry would suffer from this problem shows the worth and significance of solving it.

About distributed theories: Some theories are developed from the specific distributed system use scenarios and to improve its performance. Even with solid mathematical proof of the validity of the theory, it may often be useless at all when the theory is only based on ideal scenario that may never exist in real cases, because the theory may ignore many real factors in the distributed system when it needs to be normalized in the mathematical model. These ignorant real factors can be a strike to the feasibility of the theory because of the system complexity.

A theory from a specific system can only prove itself when it is implemented and used in that system. Even with some heuristic approaches, evaluations with real life applications witness the usefulness of the theory.

About the trends in distributed resource allocation: Despite the current super diverse types of distributed computing platforms, the resource allocation mechanism of the distributed systems will definitely go further in the fine-grained control style. The grain level from data center, cluster and node, to virtual node and resource container, the resource manager provides effective control on every level to insure efficient resource utilization, as well as resource isolation to prevent performance interference. Another important direction is the management of diverse types of devices (no matter soft or hard). Besides, CPU cores, memories and disks, the common resources such as network bandwidths, I/O ports and file handlers need to be well managed in the distributed environment.

About the relation between distributed systems and applications: Besides the conventional requirements for distributed systems, such as scalable and reliable, the distributed systems are in the first place developed to solve the various large-scale problems of different application scenarios. A solid understanding of the application use cases can always help in improving the performance of the distributed systems. At the time of being applicable to a range of applications, an ideal distributed systems should optimize itself for the target application scenarios.

Finally, about distributed machine learning: Machine learning (or deep learning) will see its brighter future in the distributed approach.

Bibliography

[1] Amazon ec2 instance types.

[2] Hku gideon-ii cluster.

[3] Movielensals spark submit 2014.

[4] Multiadjacentlist benchmark.

[5] Spark job scheduling: Dagscheduler.

[6] Spark kmeans benchmark.

[7] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, Philadelphia, PA, June 2014. USENIX Association.

[8] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012.

[9] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, volume 10, pages 19–19, 2010.

[10] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed

congestion-aware load balancing for datacenters. In *Proceedings of the ACM Conference on SIGCOMM*, pages 503–514, New York, NY, USA, 2014.

[11] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM Conference on SIGCOMM*, pages 435–446, New York, NY, USA, 2013.

[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[13] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM Conference on SIGCOMM*, volume 41, pages 242–253. ACM, 2011.

[14] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the 24th annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 193–204, 2012.

[15] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[16] Dimitri P Bertsekas and Robert G Gallager. *Data networks*, volume 2. Prentice-Hall International, 1992.

[17] Dhruba Borthakur. Hdfs architecture guide. *Hadoop Apache Project. http://hadoop. apache. org/common/docs/current/hdfs_design. pdf*, 2008.

[18] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[19] Aydin Buluc and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

[20] Thomas Cheatham, Amr Fahmy, Dan Stefanescu, and Leslie Valiant. Bulk synchronous parallel computing—a paradigm for transportable software. In *Tools and Environments for Parallel and Distributed Systems*, pages 61–76. Springer, 1996.

[21] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 390–399, 2011.

[22] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82. ACM, 2009.

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

[24] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the ACM Conference on SIGCOMM*, pages 393–406, New York, NY, USA, 2015.

[25] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *Proceedings of the ACM Conference on SIGCOMM*, 41(4):98–109, 2011.

[26] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the ACM Conference on SIGCOMM*, pages 443–454, New York, NY, USA, 2014.

[27] Jonathan Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[28] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, volume 10, page 20, 2010.

[29] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[30] Christina Delimitrou and Christos 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)*, pages 127–144, New York, NY, USA, 2014.

[31] Peter J Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 915–922. ACM, 1968.

[32] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the ACM Conference on SIGCOMM*, pages 431–442, New York, NY, USA, 2014.

[33] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[34] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 277–284. IEEE, 2008.

[35] Sylvain Gault and Christian Perez. Dynamic scheduling of mapreduce shuffle under bandwidth constraints. In *European Conference on Parallel Processing*, pages 117–128. Springer, 2014.

[36] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and

Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM Conference on SIGCOMM*, volume 39, pages 51–62, 2009.

[37] Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, and Judy Qiu. Scalable parallel computing on clouds using twister4azure iterative mapreduce. *Future Generation Computer Systems*, 29(4):1035–1048, 2013.

[38] Yanfei Guo, Jia Rao, and Xiaobo Zhou. ishuffle: Improving hadoop performance with shuffle-on-write. *10th International Conference on Autonomic Computing*, pages 107–117, 2013.

[39] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[40] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 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)*, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[41] Shadi Ibrahim, Hai Jin, Bin Cheng, Haijun Cao, Song Wu, and Li Qi. Cloudlet: towards mapreduce implementation on virtual machines. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 65–66. ACM, 2009.

[42] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, New York, NY, USA, 2007.

[43] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, pages 261–276, New York, NY, USA, 2009.

[44] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[45] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 13–24, New York, NY, USA, 2013.

[46] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *The 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–103, May 2010.

[47] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, pages 237–252, 1998.

[48] M Kiran, Ajit Kumar, and BR Prathap. Verification and validation of parallel support vector machine algorithm based on mapreduce program model on hadoop cluster. In *IEEE International Conference on Advanced Computing and Communication Systems*, pages 1–6, 2013.

[49] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *Conference on Innovative Data Systems Research (CIDR)*, volume 1, pages 2–1, 2013.

[50] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86. ACM, 2010.

[51] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[52] 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)*, pages 281–290, May 2015.

[53] Feng Liang and Francis 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*, pages 281–284. ACM, 2016.

[54] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 95–106. ACM, 2010.

[55] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[56] Huan Liu and Dan Orban. Cloud mapreduce: a mapreduce implementation on top of a cloud operating system. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 464–474. IEEE, 2011.

[57] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[58] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3):281–292, November 2014.

[59] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 135–146, New York, NY, USA, 2010.

[60] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[61] Thomas Mensink, Jakob Verbeek, Florent Perronnin, and Gabriela Csurka. Metric learning for large scale image classification: Generalizing to new classes at near-zero cost. In *Computer Vision–ECCV 2012*, pages 488–501. Springer, 2012.

[62] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM Conference on SIGCOMM*, pages 39–50, New York, NY, USA, 2009.

[63] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *6th USENIX Conference on File and Storage Technologies (FAST)*, pages 175–188, 2008.

[64] Jorda Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for mapreduce environments. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 373–380. IEEE, 2010.

[65] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: sharing the network in cloud computing. In *Proceedings of the ACM Conference on SIGCOMM*, pages 187–198, 2012.

[66] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14, 2010.

[67] Zhen Qiu, Cliff Stein, and Yuan 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)*, pages 294–303, New York, NY, USA, 2015.

[68] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Proceedings of 18th IEEE International Parallel and Distributed Processing Symposium*, pages 111–, 2004.

[69] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, November 1984.

[70] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In *Job scheduling strategies for parallel processing*, pages 110–131. Springer, 2010.

[71] Anish Das Sarma, Foto N. Afrati, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Endowment*, 6(4):277–288, 2013.

[72] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[73] Jiří Sgall. On-line scheduling. In *Online algorithms*, pages 196–231. Springer, 1998.

[74] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2011.

[75] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE, 2010.

[76] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of VLDB*, 2(2):1626–1629, 2009.

[77] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[78] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[79] Milan Vojnović, Jean-Yves Le Boudec, and Catherine Boutremans. Global fairness of additive-increase and multiplicative-decrease with heterogeneous round-trip times. In *INFOCOM*, volume 3, pages 1303–1312, 2000.

[80] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 57. ACM, 2011.

[81] Tom White. *Hadoop: The Definitive Guide, 4th Edition.* "O'Reilly Media, Inc.", 2015.

[82] Xinzhou Wu, R Srikant, and James R Perkins. Scheduling efficiency of distributed greedy scheduling algorithms in wireless networks. *Transactions on Mobile Computing*, 6(6):595–605, 2007.

[83] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[84] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[85] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 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*, pages 1–14, 2008.

[86] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[87] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 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)*, pages 2–2, 2012.

[88] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[89] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[90] Henan Zhao and Rizos Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium*, pages 14–, 2006.

[91] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 424–432, April 2015.