iShuffle: Improving Hadoop Performance with Shuffle-on-Write

Yanfei Guo, Jia Rao, and Xiaobo Zhou
Department of Computer Science
University of Colorado, Colorado Springs, USA
{*yguo,jrao,xzhou*}*@uccs.edu*

Abstract

Hadoop is a popular implementation of the MapReduce framework for running data-intensive jobs on clusters of commodity servers. Although Hadoop automatically parallelizes job execution with concurrent map and reduce tasks, we find that, *shuffle*, the all-to-all input data fetching phase in a reduce task can significantly affect job performance. We attribute the delay in job completion to the coupling of the shuffle phase and reduce tasks, which leaves the potential parallelism between multiple waves of map and reduce unexploited, fails to address data distribution skew among reduce tasks, and makes task scheduling inefficient. In this work, we propose to decouple shuffle from reduce tasks and convert it into a platform service provided by Hadoop. We present *iShuffle*, a user-transparent shuffle service that pro-actively pushes map output data to nodes via a novel *shuffle-on-write* operation and flexibly schedules reduce tasks considering workload balance. Experimental results with representative workloads show that iShuffle reduces job completion time by as much as 30.2%.

1 Introduction

Hadoop is a popular open-source implementation of the MapReduce programming model for processing large volumes of data in parallel [7]. Each job in Hadoop consists of two dependent phases, each of which contains multiple user-defined *map* or *reduce* tasks. These tasks are distributed independently onto multiple nodes for parallel execution. The decentralized execution model is essential to Hadoop's scalability to a large number of nodes as map computations can be placed near their input data stored on individual nodes and there is no communication between map tasks.

There are many existing studies focusing on improving the performance of map tasks. Because data locality is critical to map performance, work has been done to preserve locality via map scheduling [21] or input replication[4]. Others also designed interference [5] and topology [14] aware scheduling algorithms for map tasks. While there is extensive work exploiting the parallelism and improving the efficiency in map tasks, only a few studies have been devoted to expedite reduce tasks.

The all-to-all input data fetching phase in a reduce task, known as *shuffle*, involves intensive communications between nodes and can significantly delay job completion. Because the shuffle phase usually needs to copy intermediate output generated by almost all map tasks, techniques developed for improving map data locality are not applicable to reduce tasks [16, 21]. Hadoop strives to hide the latency incurred by the shuffle phase by starting reduce tasks as soon as map output files are available. There is existing work that tries to overlap shuffle with map by proactively sending map output [6] or fetching map output in a globally sorted order [19].

Unfortunately, the coupling of *shuffle* and *reduce* phases in a reduce task presents challenges to attaining high performance in Hadoop clusters and makes existing approaches [6, 19] less effective in production systems. First, in production systems with limited number of *reduce slots*, a job often executes multiple waves of reduce tasks. Because the shuffle phase starts when the corresponding reduce task is scheduled to run, only the first wave of reduce can be overlapped with map, leaving the potential parallelism unexploited. Second, tasks scheduling in Hadoop is oblivious of the data distribution skew among reduce tasks [8, 11, 12], machines running shuffle-heavy reduce tasks become bottlenecks. Finally, in a multi-user environment, one user's long-running shuffle may occupy the reduce slots that would otherwise be used more efficiently by other users, lowering the utilization and throughput of the cluster.

In this paper, we propose to decouple the *shuffle* phase from reduce tasks and convert it into a platform service provided by Hadoop. We present *iShuffle*, a user-transparent shuffle service that overlaps the data shuf-

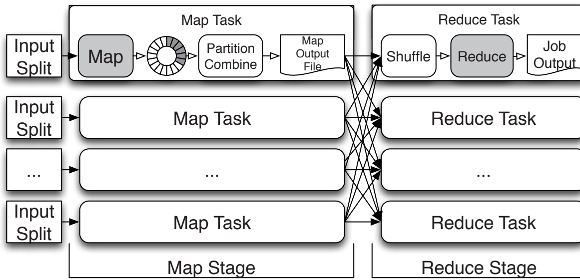


Figure 1: An overview of data processing in Hadoop MapReduce framework.

fling of any reduce task with the map phase, addresses the input data skew in reduce tasks, and enables efficient reduce scheduling. iShuffle features a number of key designs: (1) proactive and deterministic pushing shuffled data from map to Hadoop nodes when map output files are materialized to local file systems, a.k.a, *shuffle-on-write*. (2) automatic predicting reduce execution time based on the input partition size and placing the shuffled data to mitigate the *partition skew* and to avoid hotspots. (3) binding reduce tasks with data partitions only when reduce is scheduled to realize the load balancing enabled by the partition placement.

We implemented iShuffle on a 32-node Hadoop cluster and evaluated its benefits using the Purdue MapReduce Benchmark Suite (PUMA) [2] with datasets collected from real applications. We compared the performance of iShuffle running both shuffle-heavy and shuffle-light workloads with that of the stock Hadoop and a recently proposed approach (i.e., Hadoop-A in [19]). Experimental results show that iShuffle reduces job completion time by 30% and 22% compared with stock Hadoop and Hadoop-A, respectively. iShuffle also achieves significant performance gain in a multi-user environment with heterogeneous workloads.

The rest of this paper is organized as follows. Section 2 introduces the background of Hadoop, discusses existing issues, and presents a motivating example. Section 3 elaborates iShuffle's key designs. Section 4 gives the testbed setup, experimental results and analysis. Related work is presented in Section 5. We conclude this paper in Section 6.

2 Background and Motivation

2.1 Hadoop MapReduce Framework

The data processing in MapReduce [7] model is expressed as two functions: *map* and *reduce*. The map function takes an input pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key are grouped together and then passed to the same reduce function via *shuffle*, an all-map-to-all-reduce communication. The reduce function processes the intermediate key with the list of its values and generate the final results.

Hadoop's implementation of the MapReduce programming model pipelines the data processing and provides fault tolerance. Figure 1 shows an overview of job execution in Hadoop. The Hadoop runtime partitions the input data and distributes map tasks onto individual cluster nodes for parallel execution. Each map task processes a logical split of the input data that resides on the Hadoop Distributed File System (HDFS) and applies the user-defined map function on each input record. The map outputs are partitioned according to the number of reduce tasks and combined into keys with associated lists of values. A map task temporarily stores its output in a circular buffer and writes the output files to local disk every time the buffer becomes full (i.e., buffer *spill*).

A reduce task consists of two phases: *shuffle* and *reduce*. The shuffle phase fetches the map outputs associated with a reduce task from multiple nodes and merges them into one reduce input. An external merge sort algorithm is used when the intermediate data is too large to fit in memory. Finally, a reduce task applies the user-defined reduce function on the reduce input and writes the final result to HDFS. The reduce phase can not start until all the map phases have finished as the reduce function depends on the output generated by all the map tasks. To overlap the execution of map and reduce, Hadoop allows an early start of the shuffle phase (by scheduling the corresponding reduce task) as soon as 5% of the map tasks have finished.

In the next, we discuss several issues related to shuffle and reduce in the existing Hadoop framework, and give a motivating example showing how these issues affect the performance and efficiency of a Hadoop cluster.

2.2 Input Data Skew among Reduce Tasks

The output of a map task is a collection of intermediate keys and their associated value lists. Hadoop organizes each output file into partitions, one per reduce task and each containing a different subset of the intermediate key space. By default, Hadoop determines which partition a key/value pair will go to by computing a hash value. Since the intermediate output of the same key are always assigned to the same partition, skew in the input data set will result in disparity in the partition sizes. Such a *partitioning skew* is observed in many applications running in Hadoop [8, 11, 12]. Some user-defined partitioner may mitigate the skew but does not guarantee an even data distribution among reduce tasks. As a result, some reduce tasks take significant longer time to complete, slow-

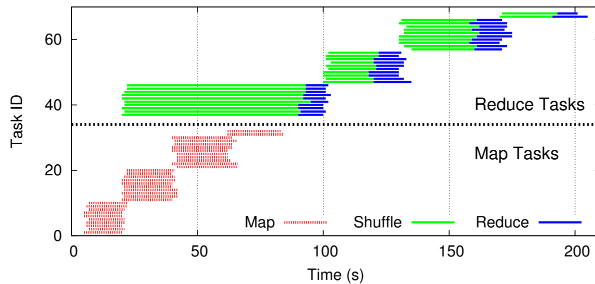


Figure 2: *tera-sort* job execution.

ing down the entire job.

2.3 Inflexible Scheduling of Reduce Tasks

Reduce tasks are created and assigned a task ID by Hadoop during the initialization of a job. The task ID is then used to identify the associated partition in each map output file. For example, shuffle fetches the partition that matches the reduce ID from all map tasks. When there are reduce slots available, reduce tasks are scheduled in the ascending order of their task IDs. Although such a design simplifies task management, it may lead to long job completion time and low cluster throughput. Due to the strict scheduling order, it is difficult to prioritize reduce tasks that are predicted to run longer than others. Further, partitions required by a reduce task may not be generated at the time it is scheduled, occupying the reduce slot and wasting cluster cycles which would otherwise be used by another reduce with all partitions ready.

2.4 Tight Coupling of Shuffle and Reduce

As part of a reduce task, shuffle can not start until the corresponding reduce is scheduled. Besides the inefficiency of job execution, the coupling of shuffle and reduce also leaves the potential parallelism between within and between jobs unexploited. In a production environment, a MapReduce cluster is shared by many users and multiple jobs [21]. Each job only gets a portion of the execution slots and often requires multiple execution waves, each of which consists of one round of map or reduce tasks. Because of the coupling, data shuffling in later reduce waves can not be overlapped with map waves.

Figure 2 shows the execution of one *tera-sort* job with 4GB dataset in a 10-node Hadoop cluster. Each node was configured with 1 map slot and 1 reduce slot. The job was divided into 32 map tasks and 32 reduce tasks [7, 17], resulting in 4 map and reduce waves. We use the duration of the shuffle phase between last execution wave and next reduce phase, termed as *shuffle delay*, to quantify how data shuffling affects the completion of reduce tasks. Due to the overlapped execution, the first reduce wave experienced a shuffle delay of 11 seconds. Unfortunately, remaining reduce waves had on average a delay of 23 seconds before the reduce phase could start. Given that the average length of the reduce phase was 25 seconds, the reduce waves would have been completed in less than half the time if the shuffle delay can be completely overlapped with map.

Figure 2 also suggests that although the overlapping of reduce and map reduced the shuffle delay from 23 to 11 seconds, the first reduce wave occupied the slots three times longer than the following waves. Most time was spent in the shuffle phase waiting for the completion of map tasks. In production systems, allowing other jobs to use these slots may outweigh the benefits brought by the overlapped execution.

These observations revealed the negative impacts of coupling shuffle and reduce on job execution and motivated us to explore a new shuffling design for Hadoop. We found that decoupling shuffle from reduce provides a number of benefits. It enables skew-aware placement of shuffled data, flexible scheduling of reduce tasks, and complete overlapping the shuffle phase with map tasks. In Section 3, we present *iShuffle*, a decoupled shuffle service for Hadoop.

3 iShuffle Design

We propose *iShuffle*, a job-independent shuffle service that pushes the map output to its designated reduce node. It decouples shuffle and reduce, and allows shuffle to be performed independently from reduce. It predicts the map output partition sizes and automatically balances the placement of map output partitions across nodes. iShuffle binds reduce IDs with partition IDs lazily at the time reduce tasks are scheduled, allowing flexible scheduling of reduce tasks.

3.1 Overview

Figure 3 shows the architecture of iShuffle. iShuffle consists of three components: *shuffler*, *shuffle manager*, and *task scheduler*. The shuffler is a background thread that collects intermediate data generated by map tasks and predicts the size of individual partitions to guide the partition placement. The shuffle manager analyses the partition sizes reported by all shufflers and decides the destination of each partition. The shuffle manager and shufflers are organized in a layered structure which is similar to Hadoop's `JobTracker` and `TaskTrackers`. The task scheduler extends existing Hadoop schedulers to support flexible scheduling of reduce tasks. We briefly describe some major features of iShuffle.

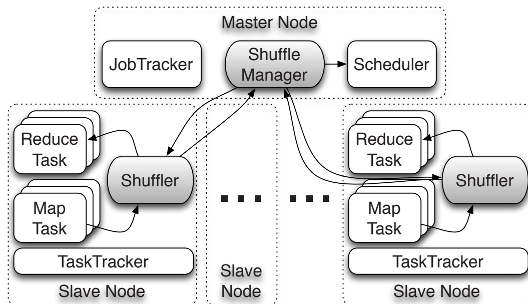


Figure 3: The architecture of iShuffle.

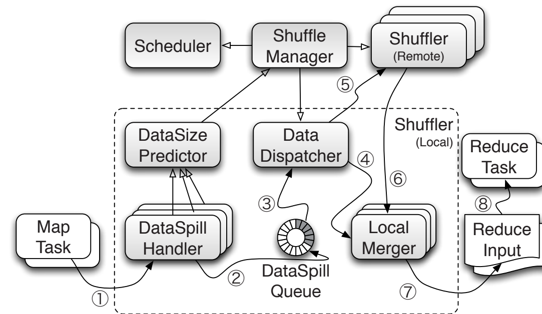


Figure 4: Workflow of Shuffle-on-Write.

User-Transparent Shuffle Service - A major requirement of iShuffle design is the compatibility to existing Hadoop jobs. To this end, we design shufflers and the shuffle manager as job-independent components, which are responsible for collecting and distributing map output data. This design allows the cluster administrator to enable or disable iShuffle through the options in the configuration files. Any user job can use iShuffle service without modifications.

Shuffle-on-Write - The shuffler implements a shuffle-on-write operation that proactively pushes the map output data to different nodes for future reduce tasks every time such data is written to local disks. The shuffling of all map output data can be performed before the execution of reduce tasks.

Automated Map Output Placement - The shuffle manager maintains a global view of partition sizes across all slave nodes. An automated partition placement algorithm is used to determine the destination for each map output partition. The objective is to balance the global data distribution and mitigate the non-uniformity reduce execution time.

Flexible Scheduling of Reduce Tasks - The task scheduler in iShuffle assigns a partition of a reduce task only when the task is dispatched to a node with available slots. To minimize reduce execution time, iShuffle always associates partitions that are already resident on the reduce node to the scheduled reduce.

3.2 Shuffle-on-Write

iShuffle decouples *shuffle* from a *reduce* task and implements data shuffling as a platform service. This allows the shuffle phase to be performed independently from map and reduce tasks. The introduction of iShuffle to the Hadoop environment presents two challenges: user transparency and fault tolerance.

Besides user-defined *map* and *reduce* functions, Hadoop allows customized *partitioner* and *combiner*. To ensure that iShuffle is user-transparent and does not require any change to the existing MapReduce jobs, we design the *Shuffler* as an independent component in the `TaskTracker`. It takes input from the combiner, the last user-defined component in map tasks, performs data shuffling and provides input data for reduce tasks. The shuffler performs data shuffling every time the output data is written to local disks by map tasks, thus we name the operation *shuffle-on-write*.

Figure 4 shows the workflow of the Shuffler. It has three stages: (1) map output collection (step ①②); (2) data shuffling (step ③④⑤⑥); (3) map output merging (step ⑦⑧).

Map output collection - The shuffler contains multiple `DataSpillHandler`, one per map task, to collect map output that has been written to local disks. Map tasks write the stored partitions to the local file system when a spill of the in-memory buffer occurs. We intercept the writer class `IFile.Writer` in the combiner and add a `DataSpillHandler` class to it. While the default writer writing a spill to local disk, the `DataSpillHandler` copies the spill to a circular buffer, `DataSpillQueue`, from where data is shuffled/dispatched to different nodes in Hadoop. During output collection, the `DataSizePredictor` monitors input data sizes and resulted partition sizes, and reports these statistics to the shuffle manager.

Data shuffling - The shuffler proactively pushes data partitions to nodes where reduce tasks will be launched. Specifically, a `DataDispatcher` reads a partition from the `DataSpillQueue` and queries the shuffle manager for its destination. Based on the placement decision, a partition could be dispatched to the shuffler on a different node or to the `local merger` in the same shuffler.

Map output merging - The map output data shuffled at different times needs to be merged to a single reduce input file and sorted by key before a reduce task can use it. The `local merger` receives remotely and locally shuffled data and merges the partitions belonging to the same reduce task into one reduce input. To ensure correctness, the `merger` only merges partitions from suc-

cessfully finished map tasks.

3.2.1 Fault Tolerance

iShuffle is robust to the failure of map and reduce tasks. Similar to [6], iShuffle maintains a bookkeeping of spill files from all map tasks. If a map task fails, its data spills in the `DataSpillQueue` and `merger` will be discarded. The `merger` merges partitions only when the corresponding map tasks commit their execution to the JobTracker. This prevents reduce tasks from using incomplete data. We also keep the merged reduce inputs in the `merger` until reduce tasks finish. In case of a failed reduce task, a new reduce can be started locally without fetching all the needed map output.

3.3 Automated Map Output Placement

The shuffle-on-write workflow relies on key information about the partition placement for each running job. The objective of partition placement is to balance the distribution of map output data across different nodes, so that the reduce workloads on different nodes are even. The optimal partition placement can be determined when the sizes of all partitions are known. However, this requires that all map tasks are finished when making the placement decisions, which effectively enforce a serialization between map tasks and the shuffle phase. iShuffle estimates the final partition sizes based on the amount of processed input data and current partition size, and uses the estimation to guide partition placement.

3.3.1 Prediction of Partition Sizes

The size of a map output partition depends on the size of its input dataset, the map function, and the partitioner. Verma *et al.* [18], found that the ratio of map output size and input size, also known as *map selectivity*, is invariant given the same job configuration. As such, the partition size can be determined using the metric of map selectivity and input data size. The shuffle manager monitors the execution of individual map tasks and estimates the map selectivity of a job by building a mathematical model between input and output sizes.

For a given job, the input dataset is divided into a number of logical splits, one per map task. Since individual map tasks run the same map function, each map task shares the same map selectivity with the overall job execution. By observing the execution of map tasks, where a number of input/output size pairs are collected, shuffle manager builds a model estimating the map selectivity metric. Shuffle manager makes k observations of the size of each map output partition. As suggested in [18], it derives a linear model between partition size and input data size:

$$p_{i,j} = a_j + b_j \cdot D_i, \tag{1}$$

where $p_{i,j}$ is the jth partition size in the ith observation and D_i is the corresponding input size. We use linear regression to obtain the parameters for m partitions, one per reduce task. Since MapReduce jobs contain many more map tasks than reduce tasks (as shown in Table 1), we are able to collect sufficient samples for building the model. Once a model is obtained, the final size of a map output partition can be calculated by replacing D_i with the actual input size of the map task.

3.3.2 Partition Placement

With predicted partition sizes, the shuffle manager determines the optimal partition placement that balances reduce workload on different nodes. Because the execution time of a reduce task is linear to its input size, evenly placing the partitions leads to balanced workload. Formally, the partition placement problem can be formulated as: given m map output partitions with sizes of $p_1, p_2, \ldots, p_m$, find the placement on n nodes, $S_1, S_2, \ldots, S_n$, that minimizes the placement difference:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\mu - \sum_{j\in S_i} p_j\right)}, \tag{2}$$

where μ is the average data size on one node.

```
Data: p: list of partition
Data: S: list of nodes, has the size of all allocated
      partitions
Result: Balanced partition placement
sort list p in descending order of partition sizes;
for i ← 1 to m do
    min_node ← S[1];
    for j ← 1 to n do
        if S[j].size < min_node.size then
            min_node ← S[j];
        end
    end
    min_node.place(p[i]);
end
```

Algorithm 1: Partition placement.

Partition placement problem can be viewed as the load balancing problem in multiprocessor systems [9] and is thus NP-hard. While the optimal solution can be prohibitively expensive to attain, we propose a heuristic-based approach to approximate an optimal placement. The detail of this approach is presented in Algorithm 1. This algorithm is based on two heuristics, the `largest partition first` for picking partitions and the `less`

Table 1: Benchmark details.

Benchmark	Input Size (GB)	Input Data	# of Maps	# of Reduce	Shuffle Volume (GB)
self-join	250	synthetic	4000	180	246
tera-sort	300	synthetic, random	4800	180	300
ranked-inverted-index	220	multi-word-count output	3520	180	235
k-means	30	Netflix data, $k = 6$	480	6	32
inverted-index	250	Wikipedia	4000	180	57
term-vector	250	Wikipedia	4000	180	59
wordcount	250	Wikipedia	4000	180	49
histogram-movies	200	Netflix data	3200	180	0.002
histogram-ratings	200	Netflix data	3200	180	0.0012
grep	250	Wikipedia	4000	180	0.0013

`workload first` for picking destination nodes. It sorts the partitions in the descending order of size and assigns the largest partition to the nodes with the least aggregate data size. It repeats until all the partitions are assigned.

3.4 Flexible Reduce Scheduling

In Hadoop, reduce tasks are assigned map output partitions statically during job initialization. When there are reduce slots available on idle nodes, reduce tasks are dispatched according to the ascending order of their task IDs. This restriction on reduce scheduling leads to inefficient execution where reduces that are waiting for map tasks to finish occupy the slots for a long time. Because iShuffle proactively pushes output partitions to nodes, it requires that reduce tasks are launched on nodes that hold the corresponding shuffled partitions. To this end, iShuffle breaks the binding of reduce tasks and map output partitions and provides flexible reduce scheduling.

An intuitive approach for flexible reduce scheduling is to traverse the task queue and find a reduce that has shuffled data on the requesting node. However, this approach does not guarantee that there is always a “local” reduce available for dispatching. iShuffle employs a different approach that assigns partitions to reduce tasks at the time of dispatching. For single-user clusters, we modified Hadoop’s FIFO scheduler to support the runtime task-partition binding. When a node with available reduce slots requests for new reduce tasks, the `task scheduler` first check with the shuffle manager to obtain the list of partitions that reside on this node. The scheduler picks the first partition in the list and associates its ID with the first reduce task in the waiting queue. The selected reduce task is then launched on the node. As such, all reduce tasks are guaranteed to have local access to their input data.

For multi-user clusters with heterogeneous workloads, we add the support for runtime task-partition association to the Hadoop Fair Scheduler (HFS). The minimum fair share allocated to individual users can negatively affect the efficiency of iShuffle as reduce tasks may be launched on remote nodes to enforce fairness. We disable such fairness enforcement for reduce tasks to support more flexible scheduling. This allows some users to temporarily run more reduce tasks than others. We rely on the following designs to preserve fairness among users and avoid starvation. First, the fair share of map tasks is still in effect, guaranteeing fair chances for users to generate map output partitions. Second, while records are sorted by key within each partition after shuffling, partitions belonging to different users are randomly placed in the list, giving each user an equal opportunity to launch reduce tasks. Finally and most importantly, reduce tasks are started only when all their input data is available. This may temporarily violates fairness, but prevents wasted cluster cycles spent in waiting for unfinished maps and results in more efficient job execution.

4 Evaluation

4.1 Testbed Setup

Our testbed was a 32-node Hadoop cluster. Each node had one 2.4 GHz 4-core Intel Xeon E5530 processor and 4 GB memory. All nodes were interconnected by a Gigabit Ethernet. The operating system uses Linux kernel 2.6.24. We deployed Hadoop stable release version 1.1.1 and each machine ran Ubuntu Linux with kernel 2.6.24. Two nodes were configured as the `JobTracker` and `NameNode`, respectively. The rest 30 nodes were configured as slave nodes for HDFS storage and MapReduce task execution. We set the HDFS block size to its default value 64 MB. Each slave node was configured with 4 map slots and 2 reduce slots, resulting in a total capacity of running 120 map and 60 reduce tasks simultaneously

in the cluster.

For comparison, we also implemented Hadoop-A proposed in [19]. It enables reduce tasks to access map output files on remote disks through the network. By using a priority queue-based merge sort algorithm, Hadoop-A eliminates repetitive merge and disk accesses, and removes the serialization between the shuffle and reduce phases. However, Hadoop-A requires the remote direct memory access (RDMA) feature on Infiniband interconnections for fast remote disk access. We implemented Hadoop-A using remote procedure calls on our testbed with Gigabit Ethernet and compared its performance with iShuffle on commodity hardware.

4.2 Workloads

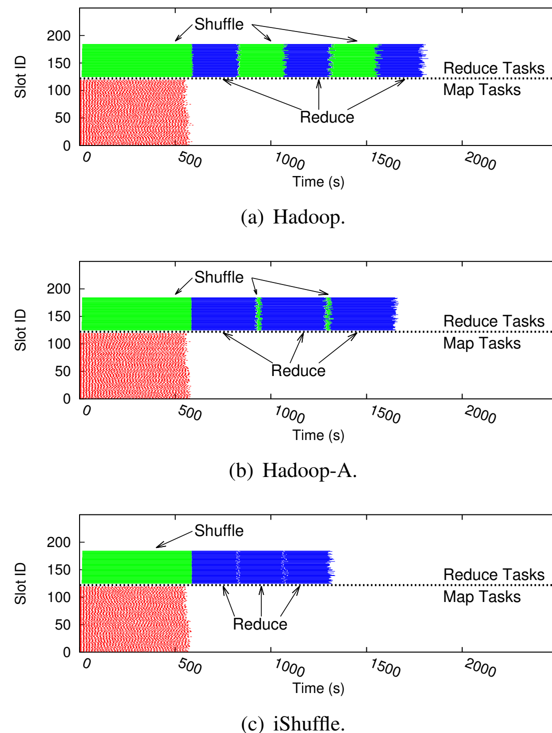
We used the Purdue MapReduce Benchmark Suite (PUMA) [2] to compose workloads for evaluation. PUMA contains various MapReduce benchmarks and real-world test inputs. Table 1 shows the benchmarks and their configurations used in our experiments. For most of the benchmarks, the number of reduce tasks was set to 180 to allow multiple reduce waves. The only exception was *k-means*, which ran on a 30 GB dataset with 6 reduce tasks.

These benchmarks can be divided into two categories: shuffle-heavy and shuffle-light. Shuffle-heavy benchmarks have high map selectivity and generate a large volume of data to be exchanged between map and reduce. Thus, such benchmarks are sensitive to optimizations on the shuffle phase. For shuffle-light benchmarks, there is little data that needs to be shuffled. We used both benchmark types to evaluate the effectiveness of iShuffle and its overhead on workloads with little communications.

4.3 Reducing Shuffle Delay

Recall that we defined shuffle delay as the duration between the last wave of execution and the next reduce wave. Shuffle delay measures the shuffle period that can not be overlapped with the previous wave. The smaller the shuffle delay, the more efficient the shuffling scheme. We ran *tera-sort* on stock Hadoop, Hadoop-A and iShuffle, and recorded the start and completion times of each map, shuffle and reduce phase.

Figure 5 shows the trace of the *tera-sort* job execution under different approaches. The X-axis is the time span of job execution and Y-axis represents the map and reduce slots. The results show that iShuffle had the best performance with 30.2% and 21.9% shorter job execution time than stock Hadoop and Hadoop-A, respectively. As shown in Figure 5(a), there is a significant delay of the reduce phase for every reduce task in stock Hadoop. Due



(a) Hadoop.

(b) Hadoop-A.

(c) iShuffle.

Figure 5: Execution trace of *tera-sort* using stock Hadoop, Hadoop-A, and iShuffle approaches.

to proactive placement of map output partitions, iShuffle had almost no shuffle delays. Note that Hadoop-A also significantly reduced shuffle delay because it operates on globally sorted partitions and can greatly overlap the shuffle and reduce phase.

iShuffle outperformed Hadoop-A on our testbed for two reasons. First, the building of the priority queue poses extra delay, e.g., the shuffle delay before the second and third reduce waves in Hadoop-A, to each reduce task. Second, the remote disk access in an Ethernet environment is significant slower than that in an Infiniband network, which leads to much longer reduce phases in Hadoop-A.

4.4 Reducing Job Completion Time

We study the effectiveness of iShuffle in reducing overall job completion time with more comprehensive benchmarks. We use the job completion time in stock Hadoop implementation as the baseline and compare the normalized performance of iShuffle and Hadoop-A. Figure 6 shows the normalized job completion time of all benchmarks listed in Table 1. The results show that for shuffle-heavy benchmarks such as *self-join*, *tera-sort*, and *ranked-inverted-index*, iShuffle outperformed the stock Hadoop by 29.1%, 30.1%, and 27.5%, respec-

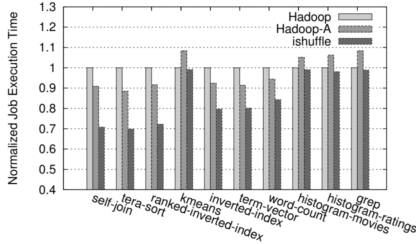


Figure 6: Job completion time using three different approaches.

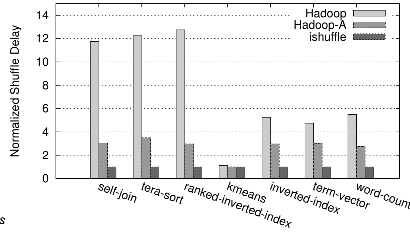


Figure 7: Shuffle delay due to three different approaches.

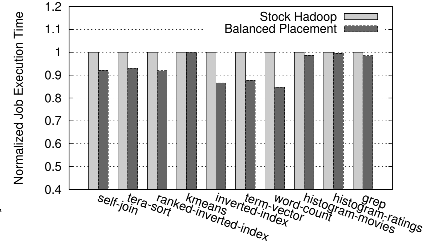


Figure 8: Performance with automated map output placement.

tively. iShuffle also outperformed Hadoop-A by 22.7%, 21.9%, and 21.1% in these benchmarks. The result with *k-means* benchmark does not show significant job execution time reduction between iShuffle and original Hadoop. This is because *k-means* only has 6 reduce tasks. With only one wave of reduce tasks, stock Hadoop was able to overlap the shuffle phase with map tasks and had similar performance as iShuffle. However, due to the additional delay of remote disk access, Hadoop-A had longer reduces, thus longer overall completion time.

Benchmarks like *inverted-index*, *term-vector*, and *wordcount* also fit in the shuffle-heavy category, but the shuffle volumes are smaller than other shuffle-heavy benchmarks. These benchmarks had less shuffle delay than other shuffle-heavy benchmarks simply because there was less data to be copied during the shuffle phase. Therefore, the performance improvement due to iShuffle was less. Figure 6 shows that iShuffle achieved 20.3%, 19.7%, and 15.6% better performance than stock Hadoop with these benchmarks, respectively. For these benchmarks, Hadoop-A still gained some performance improvement over stock Hadoop as the reduction on shuffle delay outweighed the prolonged reduce phase. However, the performance gain was marginal with 7.5%, 8.6%, and 5.5% improvement, respectively.

For the shuffle-light benchmarks, because the shuffle delay is negligible. Both iShuffle and Hadoop-A achieves almost no performance improvement. The performance degradation due to remote disk access in Hadoop-A is more obvious in this scenario.

We also compare the shuffle delay between the stock Hadoop, iShuffle, and Hadoop-A. Figure 7 shows the comparison of normalized shuffle delay. We used the shuffle delay of iShuffle as the based line. The results agree with the observation we made in previous experiments. iShuffle was able to reduce the shuffle delay significantly if the job had large volumes of shuffled data and multiple reduce waves. For benchmarks that have the largest shuffle-volume, the reductions in shuffle delay were more than 10x compared with stock Hadoop. For benchmarks with medium shuffle volume, the improvement on shuffle delay was from 4.5x to 5.5x. Figure 7 also suggests that iShuffle was on average 2x more effective in reducing shuffle delay than Hadoop-A.

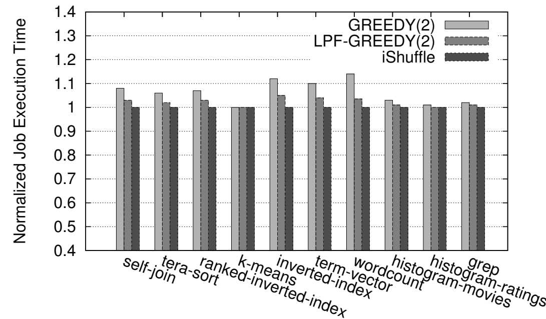


Figure 9: Performance with different placement balancing algorithms.

4.5 Balanced Partition Placement

We have shown that iShuffle effectively hides shuffle latency by overlapping map tasks and data shuffling. In this subsection, we study how the balanced partition placement affects job performance. To isolate the effect of partition placement, we first ran benchmarks under stock Hadoop and recorded dispatching history of reduce tasks. Then, we configured iShuffle to place partitions on nodes in a way that leads to the same reduce execution sequence. As such, job execution enjoys overlapped shuffle provided by iShuffle, but bears the same partitioning skew in stock Hadoop. We compare the performance with balanced partition placement and stock Hadoop.

Figure 8 shows the performance improvement due to balanced partition placement. The results show that iShuffle achieved 8-12% performance improvement over stock Hadoop. We attribute the performance gain to the prediction-based partition placement that mitigates the partitioning skew. It prevents straggler tasks from prolonging job execution time. The partition placement in iShuffle relies on accurate predictions of the individual

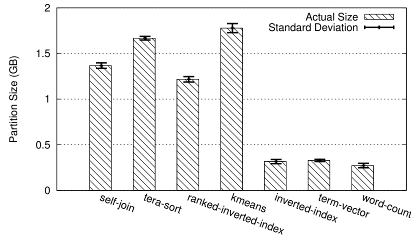


Figure 10: Accuracy of iShuffle partition size prediction.

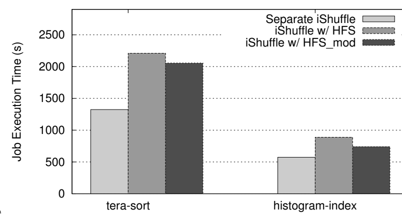


Figure 11: Performance of job mix of *tera-sort* and *histogram-movies*.

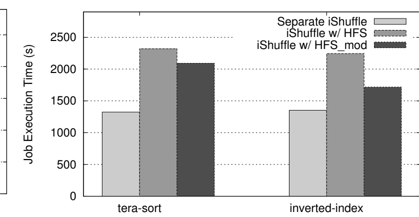


Figure 12: Performance of job mix of *tera-sort* and *inverted-index*.

partition sizes. Figure 10 shows the differences between measured partition sizes and the predicted ones. The results suggest that for all the shuffle-heavy benchmarks, iShuffle was able to estimate the final partition size with no more than 2% prediction errors.

4.6 Different Balancing Algorithms

In this subsection, we study how different partition balancing algorithms affect job performance. We compare our heuristic based partition balancing algorithm with two representative randomized balancing approaches.

`GREEDY(2)` implements the two-choice randomized load balancing algorithm proposed in [15]. It randomly picks up a map task for output placement and makes decision on which slave node to place the output using the two-choice greedy algorithm (i.e., `GREEDY(2)`). The node with less aggregated partition size (breaking ties arbitrarily) in the two randomly picked nodes is selected as the destination for the output placement. Different from `GREEDY(2)` which selects tasks randomly for placement, `LPF-GREEDY(2)` sorts tasks according to the descending order of their predicted partition sizes and always places tasks with larger partitions first (i.e., *largest partition first (LPF)*). Node selection is based on the two-choice randomized strategy.

Figure 9 compares the performance of different balancing algorithms. The results show that the simple heuristics used in iShuffle achieved $8-12\%$ shorter job completion time than `GREEDY(2)` in shuffle-heavy workloads (e.g., inverted-index). Since balanced partition placement is critical to job performance, `GREEDY(2)`'s randomization in task selection made it difficult to evenly distribute computation across nodes and contributed to the prolonged execution times. To confirm this, we ran `LPF-GREEDY(2)` with the same set of workloads. With the *largest partition first* heuristic in task selection, `LPF-GREEDY(2)` achieved close performance (on average only 2.5% longer runtimes) to iShuffle. In summary, although randomized balancing algorithms are easy to implement, the heuristics use in iShuffle is key to achieving balanced output placement.

4.7 Running Multiple Jobs

We further evaluate iShuffle in a multi-user Hadoop environment. We created multiple workload mixes, each contained two different MapReduce jobs. We ran one workload at a time with two jobs sharing the Hadoop cluster. We modified the Hadoop Fair Scheduler (HFS) (i.e., *iShuffle w/ HFS_mod*) to support runtime task-partition binding. For comparison, we also study the performance of iShuffle with the original HFS that enforces a minimum fair share on reduce tasks (i.e., *iShuffle w/ HFS*) and iShuffle running a single job on a dedicated cluster (i.e., *Separate iShuffle*).

The first experiment used the combination of a shuffle-heavy job and a shuffle-light job. Figure 11 shows the result of workload mix of *tera-sort* and *histogram-movies*. The results suggest that the modified HFS outperformed the original HFS by 16% and 25% for *tera-sort* and *histogram-movies*, respectively. Unlike the original HFS, which guarantees *max-min* fairness to jobs, iShuffle allows the reduce of one job to use more reduce slots. iShuffle prioritizes shuffle-light jobs because the execution time of their reduce tasks is short. Allowing shuffle-light jobs to run with more slots boosted their performance significantly. Although shuffle-heavy jobs suffered unfairness to a certain degree, their overall performance under the modified HFS was still better than that under the original HFS.

Next, we perform the experiment with two shuffle-heavy jobs. Figure 12 shows the performance of *tera-sort* and *inverted-index*. It shows that iShuffle improved job execution times by 8% and 23% over the original HFS in these two benchmarks. Although the size of input datasets of these two benchmarks are similar, *inverted-index* has a smaller shuffle volume. Therefore, its reduce tasks can be started earlier as their partitions required less time to shuffle. *tera-sort* had less improvement in this scenario because some of its reduce tasks are delayed by *inverted-index*. Table 2 shows more results of iShuf-

Table 2: Job completion time of co-running jobs.

Workload Mix		Stock Hadoop		iShuffle	
A +	B	A	B	A	B
tera-sort+	grep	2210	1247	2144	1038
tera-sort+	histogram -ratings	2308	653	1976	530
tera-sort+	term-vector	2576	2183	2349	1845
tera-sort+	wordcount	2341	1433	2126	1197
tera-sort+	k-means	1723	3764	3685	3748

fle with heterogeneous workloads compared with stock Hadoop. For most workload mixes with two jobs, iShuffle w/ modified HFS was able to reduce the job completion time for both jobs. The performance gain depends on the amount of shuffled data in these co-running jobs.

However, iShuffle had poor performance with workload mix *tera-sort* + *k-means*. We ran *tera-sort* with a 300GB dataset and *k-means* with a 15GB dataset. The result of *k-means* does not agree with previous observations for shuffle-light workloads. The co-running of *tera-sort* and *k-means* significantly degraded the performance of *tera-sort*. An examination of the execution trace revealed that although *k-means* has little data to exchange between map and reduce, it is compute intensive. iShuffle started *k-means* earlier than *tera-sort* and *k-means* occupied the reduce slots for a long time delaying the execution of *tera-sort*. The culprit was that for *k-means*, the partition size is not a good indicator of the execution time of its reduce tasks. Thus, iShuffle failed to balance the reduce workload on multiple nodes. A possible solution is to detect such outliers earlier and restart them on different nodes. Since such outliers often have small shuffle volume, the migration is not expensive.

5 Related Work

MapReduce is a programming model for large-scale data processing [7]. Hadoop, the open-source implementation of MapReduce, provides a software framework to support the distributed processing of large datasets [1].

There is great research interest in improving Hadoop from different perspectives. A rich set of research focused on the performance and efficiency of Hadoop cluster. Jiang *et al.* [10], conducted a comprehensive performance study of Hadoop, summarized the factors that can significantly improve the Hadoop performance. Verma *et al.* [17, 18], proposed cluster resource allocation approach for Hadoop. They focused on improving the cluster efficiency by minimizing resource allocations to jobs while maintaining their service level objectives. They estimated the execution time of a job based on its resource allocation and input dataset, and determined the minimum resource allocation for the job. Lama and Zhou [13] proposed and developed AROMA, a system that automates the allocation of Cloud resources and configuration of Hadoop parameters for achieving quality of service goals while minimizing the incurred cost. It uses a SVM-based approach to obtain the optimal job configuration. It adapts to ad-hoc jobs by robustly matching their resource utilization signature with previously executed jobs and making provisioning decisions accordingly.

A number of studies proposed different task scheduling algorithms to improve Hadoop performance. The Longest Approximate Time to End (LATE) scheduling algorithm [22] improved the job performance in heterogeneous environments. FLEX [20] is a scheduling algorithm that enforces fairness between multiple jobs in a Hadoop cluster. It optimized the performance of each job under different metrics. Zaharia *et al.*, proposed delay scheduling [21] as an enhancement to Hadoop Fair Scheduler. It exploited data locality of map task and significantly improved performance.

There are a few studies on skew mitigations. SkewReduce [11] alleviated the computational skew problem by applying a user-defined cost function on the input records. Partitioning across nodes relies on this cost function to optimize the data distribution. SkewTune [12] proposed a framework for skew mitigation. It repartitioned the long tasks to take the advantage of idle slots freed by short tasks. However, moving repartitioned data to idle nodes requires extra I/O operations.

Some recent work focused on the improvement of shuffle and reduce. MapReduce Online [6] proposed a push-based shuffle mechanism to support the online aggregation and continuous queries. MaRCO [3] overlaps the reduce and shuffle. But the early start of reduce generates partial reduces which could be the source of overhead for some applications. Hadoop Acceleration [19] proposed a different approach to mitigate shuffle delay and repetitive merges in Hadoop. It implemented a merge algorithm based on remote disk access and eliminated the explicit copying process in shuffle. However, this approach relies on the RDMA feature of Infiniband network, which is not available on commodity network hardware. Without RDMA, the remote disk access added significant overhead to reduce tasks. Moreover, Hadoop-A does not decouple shuffle and reduce, making it less effective for jobs with multiple reduce waves.

6 Conclusions

Hadoop provides a simplified implementation of the MapReduce framework, but its design poses challenges to attain the best performance in job execution due to tightly coupled shuffle and reduce, partitioning skew, and inflexible scheduling. In this paper, we propose *iShuffle*,

a novel user-transparent shuffle service that provides optimized data shuffling to improve job performance. It decouples shuffle from reduce tasks and proactively pushes data to be shuffled to Hadoop node via a novel *shuffle-on-write* operation in map tasks. iShuffle further optimizes the scheduling of reduce tasks by automatic balancing workload on multiple nodes and runtime flexible reduce scheduling. We implemented iShuffle as a configurable plug-in in Hadoop and evaluated its effectiveness on a 32-node cluster with various workloads. Experimental results shows that iShuffle is able to reduce job completion time by as much as 30.2%. iShuffle also significantly improves job performance in a multi-user Hadoop cluster running heterogeneous workloads.

Acknowledgement

This research was supported in part by U.S. NSF CAREER Award CNS-0844983 and research grant CNS-1217979.

References

[1] Apache Hadoop Project. `http://hadoop.apache.org`.

[2] PUMA: Purdue mapreduce benchmark suite. `http://web.ics.purdue.edu/~fahmad/benchmarks.htm`.

[3] AHMAD, F., LEE, S., THOTTETHODI, M., AND VIJAYKUMAR, T. N. [inpress]mapreduce with communication overlap (marco). *Journal of Parallel and Distributed Computing* (2012).

[4] ANANTHANARAYANAN, G., AGARWAL, S., KANDULA, S., GREENBERG, A., STOICA, I., HARLAN, D., AND HARRIS, E. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)* (2011).

[5] CHIANG, R. C., AND HUANG, H. H. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011).

[6] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. Mapreduce online. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2010).

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proc. of the USENIX Symposium on Operating System Design and Implementation (OSDI)* (2004).

[8] DEWITT, D., AND GRAY, J. Parallel database systems: the future of high performance database systems. *Communication of ACM 35*, 6 (1992), 85–98.

[9] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1990.

[10] JIANG, D., OOI, B. C., SHI, L., AND WU, S. The performance of MapReduce: an in-depth study. *Proc. VLDB Endow.* (2010).

[11] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the ACM Symposium on Cloud Computing (SOCC)* (2010).

[12] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. Skewtune: Mitigating skew in mapreduce applications. In *Proc. of the ACM SIGMOD* (2012).

[13] LAMA, P., AND ZHOU, X. AROMA: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC)* (2012), pp. 63–72.

[14] LI, M., SUBHRAVETI, D., BUTT, A. R., KHASYMSKI, A., AND SARKAR, P. Cam: a topology aware minimum cost flow based resource manager for mapreduce applications in the cloud. In *Proc. of the ACM Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2012).

[15] MITZENMACHER, M. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California, Berkeley, 1996.

[16] TAN, J., MENG, X., AND ZHANG, L. Coupling scheduler for mapreduce/hadoop. In *Proc. of the ACM Int'l Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2012).

[17] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. Aria: automatic resource inference and allocation for mapreduce environments. In *Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC)* (2011).

[18] VERMA, A., CHERKASOVA, L., AND CAMPBELL, R. H. Resource provisioning framework for mapreduce jobs with performance goals. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware* (2011).

[19] WANG, Y., QUE, X., YU, W., GOLDENBERG, D., AND SEHGAL, D. Hadoop acceleration through network levitated merge. In *Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011).

[20] WOLF, J., RAJAN, D., HILDRUM, K., KHANDEKAR, R., KUMAR, V., PAREKH, S., WU, K.-L., AND BALMIN, A. Flex: a slot allocation scheduling optimizer for mapreduce workloads. In *Proc. of the ACM/IFIP/USENIX Int'l Conference on Middleware* (2010).

[21] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)* (2010).

[22] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving mapreduce performance in heterogeneous environments. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).