Efficient Movement and Task Management in MapReduce for Fast Analytics of Big Data

by

Yandong Wang, B.S., M.S.

A dissertation proposal submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 4, 2014

Keywords: MapReduce, Big Data, Hadoop Acceleration, Preemptive ReduceTask,
High-Performance Computing

Approved by

Weikuan Yu, Associate Professor of Computer Science and Software Engineering
David Umphress, Associate Professor of Computer Science and Software Engineering
Dean Hendrix, Associate Professor of Computer Science and Software Engineering

Abstract

MapReduce programming model has achieved great success over the past decade. With its recognized merits such as superior scalability and strong fault tolerance, MapReduce has thrived as a primary processing engine adopted by leading enterprises for analyzing gigantic datasets and driving cloud services. Recently, in order for companies to pursue high quality-of-service and fulfill requests from many customers, the demand for enhancing MapReduce frameworks so that they can leverage the best performance from underlying systems and support multi-tenancy is growing. However, many challenges exist in optimizing contemporary MapReduce frameworks to deliver fast job completion, fairness among many users, high cluster utilization, as well as platform-aware adaptability.

This dissertation focuses on pushing forward the evolution of contemporary MapReduce frameworks. We have comprehensively analyzed several major MapReduce systems on different platforms, identified their limitations, and explored various optimization techniques to enhance their performance. In particular, this dissertation aims to address three major challenges that prevent MapReduce frameworks from achieving the optimal performance. These three challenges include (1) exploiting the design of a high-performance I/O services for accelerating the intermediate data movement for MapReduce frameworks; (2) enhancing task management to provision ideal quality-of-service in terms of efficiency and fairness in multi-tenant MapReduce clusters; (3) improving the adaptability of MapReduce frameworks for platforms featuring high performance characteristics. To address these challenges, in this dissertation, we have introduced a novel Network Levitated Merge algorithm, along with a Hadoop Acceleration framework for MapReduce to provide a high-performance I/O layer. Built on top of these techniques, MapReduce can efficiently move a deluge amount of intermediate data among a large number of nodes and yield effective

performance improvement than the otherwise. In addition, to provision satisfactory quality-of-service, this dissertation has also designed a lightweight work-conserving Preemptive ReduceTask and Fast Completion Scheduler to enhance the task management so that MapReduce can deliver both fast job execution and fairness for multi-tenant MapReduce clusters. Our evaluation with a diverse collection of workloads adequately demonstrate that our solutions can efficiently outperform the state-of-the-art MapReduce schedulers. Furthermore, to cope with the demand for leveraging MapReduce frameworks to process gigantic simulation results from scientific applications, this dissertation has thoroughly characterized MapReduce frameworks on High-Performance Computing (HPC) systems. Based on our findings, we have concluded that existing MapReduce frameworks lack the capability to fully exploit the advantages of high-performance computing facilities. Accordingly, we have introduced several enhancement techniques to optimize the adaptability of MapReduce for these platforms. Our performance examination sufficiently corroborates the effectiveness of our techniques.

Through systematic experiments and comprehensive evaluation and analysis described in this dissertation, we have demonstrated the efficacy of our innovations, meanwhile revealing many optimization spaces as well as opportunities for future research on MapReduce frameworks.

Keywords: *MapReduce, Big Data, Distributed Computing, Preemptive Scheduling*

Acknowledgments

Foremost, I gratefully thank my advisor, Dr. Weikuan Yu, for his support, patience, encouragement, and cultivation during my Ph.D study. Never would I have achieved what I have so far had I not been under his guidance. His assiduous attitude toward research, limitless fantastic ideas, and sincere while determined opinions significantly help me shape my research and go through the tough times. He has been a most valued friend and mentor in my career. I am greatly indebted for the time and efforts he spent for nurturing me as a researcher in the computer science.

In addition, I gratefully acknowledge the substantial support, guidance, and friendship I received from Dr. Li Zhang, Dr. Jian Tan and Dr. Xiaoqiao Meng during my summer internship at IBM T. J Waton research center. I am also indebted to my mentor, Ms. Robin Goldstone, for her support during my internship in Lawrence Livermore National Laboratory. I would also like to thank Dr. Sarp Oral for his guidance during my summer internship in Oak Ridge National Laboratory. Without them, my career growth would not have reached this far.

I would also like to thank my committee members, Dr. David Umphress, and Dr. Dean Hendrix for their patience and their comments on this dissertation.

I would also like to acknowledge the generous help I received from my fellow students in The Parallel Architecture and System Research Lab (PASL) at Auburn University. Special thanks go to Dr. Xinyu Que for his significant help on the Hadoop Acceleration project, Cong Xu for his help on the JVM-bypass library, Xiaobing Li for his collaboration on coordinated task management. I also appreciate the help and friendship I received from Dr. Yuan Tian, Zhuo Liu and his wife Lin Cao, Bin Wang and his wife Zixin Lou, Cong Xu and his friend Ruijuan Li, Teng Wang, Huansong, and Fang Zhou. Because of them, Auburn is like a home to me.

My deepest gratitude goes to my grandma Fuqing Liu, my father Quoqiang Wang, my mother Mingxia Deng, and my aunt Mingying Deng. Without their love and sacrifice, I would never

be here in the first place. Their love light up my path and encourage me to pursue my dream. Eventually, I want to say thank you to my wife Ziwen Sun for her endless love in my life. Thank you for holding my hands and giving me courage throughout this long journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

MapReduce programming model [33], introduced by Google, has now been the *de facto* processing engine for analyzing massive scale datasets. Nowadays, various implementations of MapReduce, including Hadoop [3], Dyrad [64], Spark [103], MR-MPI [12], and M3R [78] have obtained broad attention from both industry and academia. For example, Hadoop [3], an open-source implementation of MapReduce, has been recently endorsed by many leading IT companies, including IBM, Intel, Amazon, Yahoo!, *etc*, and applied by different organizations to support numerous challenging applications, such as large-scale graph processing [2], genomics computation [6], mining massive astrophysical datasets [74] as well as facial similarity and recognition [52]. According to the 2011 IDC report [17], the market of MapReduce and its ecosystems will continue expanding and become a multi-billion business by 2016.

Three superior characteristics of MapReduce have greatly contributed to its success. First of all, MapReduce programming model intelligently exposes simple *map* and *reduce* interfaces to application developers, meanwhile hiding the complexities, such as fault-handling, intermediate data shuffling, *etc*. Therefore, it substantially relieves the development burden from the system designers who can then focus on business logic to satisfy the needs from customers. Secondly, MapReduce aims to be *fault-tolerant*. Recognizing that failure is the norm, MapReduce frameworks assume that the underlying systems are unreliable, thus has enabled fault handling throughout the entire system design. From a high-level perspective, contemporary MapReduce systems generally rely on fine-grained job partitioning and task failure-over to resist common failure scenarios. Meanwhile, due to inherent task independence, MapReduce can avoid expensive job abortions when failures occur. Thirdly, via minimizing the coupling among tasks, MapReduce programming model exhibits superior *scalability*. A recent report [19] from Yahoo!, showing the deployment of

YARN MapReduce [13] across 30,000 nodes is the best testaments of such characteristic. Under such unprecedented scale, MapReduce applications are empowered to harness the computation capability of large clusters and address many challenges that will open up new horizons for making more discoveries.

Following the widespread popularity of MapReduce is the growing demand to further optimize its performance for various purposes. In particular, accelerating the data movement and enhancing task management are becoming critical since they fundamentally determine whether the frameworks can achieve the performance goals in different use cases. Diverting from the initial design goal that MapReduce is mainly used for batch jobs, contemporary MapReduce frameworks are being broadly employed to process interactive queries [84, 65] and streaming data [105] on platforms featuring distinct performance characteristics, including multi-tenant environments and High-Performance Computing (HPC) clusters. However, these use cases require MapReduce to possess the capabilities to quickly circulate data and provision fairness via efficiently utilizing clusters so that the latency of interactive queries can be minimized and no user or job is penalized due to unfair treatment.

In response to these challenges, the focus of this dissertation is on investigating the obstacles and opportunities to simultaneously optimize MapReduce frameworks from many sides, including speeding up single job execution, balancing fairness among concurrent workloads, meanwhile improving cluster utilization. It aims to achieve this goal via shedding light on the inefficiencies and bottlenecks in MapReduce frameworks, designing noval algorithms to exploit the performance from underlying infrastructures and balance the tradeoff among multiple performance objectives, respectively. In particular, this dissertation introduces several techniques to accelerate the intermediate data movement and enhance fairness within task management. In the rest of this chapter, we provide high-level description of the background of MapReduce frameworks and the challenges involved in the optimization, then we discuss our research contributions.

1.1 Overview of MapReduce Architectures

In this section, we describe the MapReduce programming model, and several implementations of MapReduce to help better comprehend the challenges. Given that Hadoop [3] and Spark [103] are two of the most recognized MapReduce implementations, this dissertation concentrates on optimizing these two frameworks from the perspective of intermediate data movement and task management. While the techniques are also applicable to other MapReduce implementations.

1.1.1 Data Movement in MapReduce

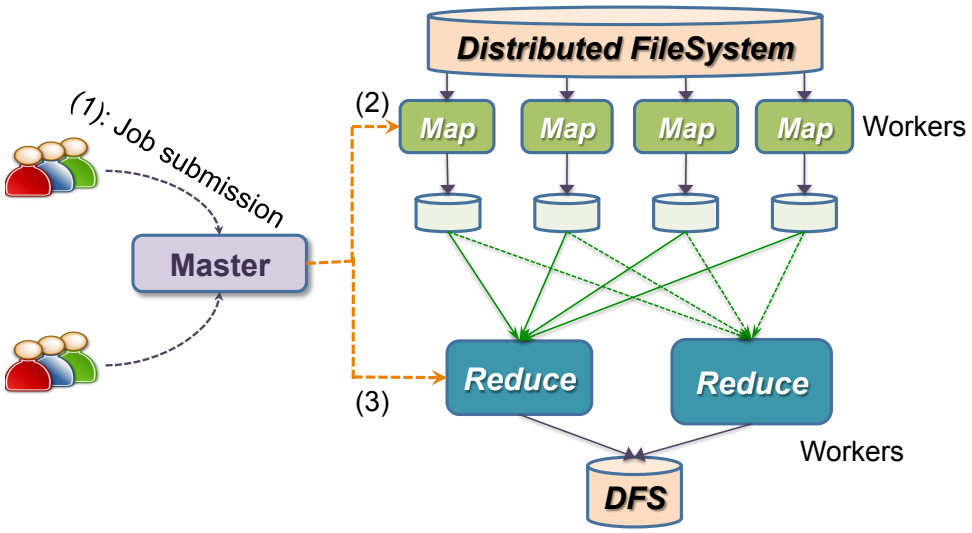


Figure 1.1: An Overview of Data Processing in MapReduce framework

Efficient data movement is critical for MapReduce frameworks. A general MapReduce framework consists of two categories of components to facilitate the data movement: a *Master*, and many *Workers*, as shown in the Figure 1.1. After users submit jobs to the Master, the Master commands Workers to process data in parallel through two main functions: map and reduce. Across this process, the Master is in charge of scheduling map tasks and reduce tasks from concurrently running jobs to Workers via following certain scheduling policies, such as First-In-First-Out (FIFO) or Fair Sharing. Meanwhile, it also monitors their progresses, collects run-time execution statistics, and handles possible faults and errors through task re-execution.

From the view of pipelined data processing, a MapReduce job needs to complete three major execution phases: *map*, *shuffle/merge*, and *reduce*, respectively. In the map phase, the Master selects a number of Workers and schedules them to run the map function. Each Worker launches several map tasks, one per split of data that is retrieved from the distributed file system, such as Hadoop Distributed File System [79] or Google File System [37, 31]. In each split, user data is organized as many records of <key,val> pairs. The mapping function in a map task converts the original records into intermediate results, which are data records in the form of <key',val'> pairs. These new data records are stored as a map output file stored on local file system, such as ext3.

In the shuffle/merge phase, when some map outputs become available, the Master selects another set of Workers to run reduce tasks. Each reduce task starts by fetching a partition that is intended for it from a map output file (also called segment). Typically, there is one segment in each map output for every reduce task. So, a reduce task needs to fetch such segments from all map output files. Globally, these fetch operations lead to an all-to-all shuffling of data segments among all the map and reduce tasks. While the data segments are being shuffled, they are also merged within each reduce task based on the order of keys in the data records, As more remote segments are fetched and merged locally, a reduce task has to spill, i.e., store, some segments to local disks in order to alleviate memory pressure. Due to the concurrent execution nature of shuffle and merge, this stage is also commonly referred as the shuffle/merge phase.

In the reduce phase, each reduce task loads and processes the merged segments using the reduce function. The final result is then stored back to the distributed file system. Although several MapReduce implementations [103, 78] allow caching the final output in the memory, we deem them as implementation details.

Hadoop [3] is an open-source implementation of MapReduce, Its design bears strong similarity to original MapReduce as described above with different names for major components: a JobTracker, *a.k.a.* Master and many TaskTrackers, *a.k.a.* Workers. The JobTracker commands TaskTrackers to process data in parallel through MapTasks and ReduceTasks, following the same data processing pipeline depicted above.

1.1.2 Task Management in MapReduce

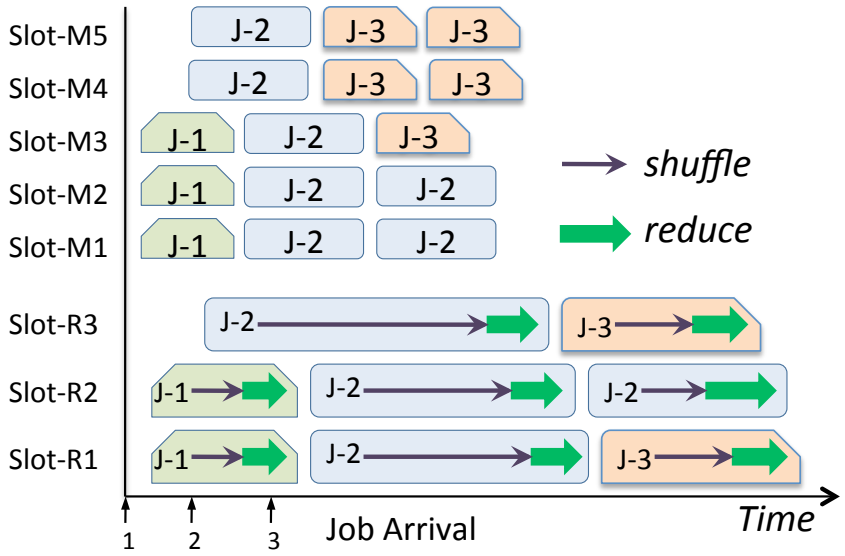


Figure 1.2: An Example of Managing Slots among Jobs

Fairness has become a primary performance goals that MapReduce schedulers strive to optimize [102, 38]. Taken Hadoop Fair scheduler as an example, the JobTracker assigns available map and reduce slots separately to jobs in the queue, one slot per task. Figure 1.2 shows an example of scheduling three jobs (represented by shaped blocks in three colors) on a system with three reduce slots and five map slots. A job when running alone can satisfy its needs with all reduce slots, but it has to share the slots when other jobs arrive. Once granted a slot, a ReduceTask has to fetch data produced by all MapTasks before it completes. In the figure, Job 1 first arrives by itself. It grabs 3 map slots and 2 reduce slots for itself and completes execution. Job 2 then takes the rest of map and reduce slots. When Job 2 needs more map or reduce slots, it has to share, because Job 3 has arrived.

Current MapReduce schedulers greedily launch as many ReduceTasks as permitted for each job to maximize the chance of overlapping the shuffling of available intermediate data with remaining map phase. However, such design can lead to monopolization behavior we will illustrate in Chapter 2. Although Hadoop allows an option (*slowstart.completed.maps*) in configuration to

delay the launch of ReduceTasks so that the small jobs after the large jobs can have chances to share the reduce slots, it does not address the issue.

1.1.3 Memory-Resident MapReduce

Spark is another highly popular MapReduce implementation introduced by UC Berkeley [5]. Recently, it has also gained broad attention from scientists at the leadership computing facilities as a promising solution to analyze gigantic simulation results. Similar to original MapReduce, it consists of two categories of components: a scheduler and many executors. The scheduler is in charge of scheduling tasks, monitoring their progress, and fault handling through task re-execution. The executors are responsible for executing the actual computing and data processing tasks. As many MapReduce implementations, Spark usually works together with distributed file systems that are designed to co-locate the storage resource (*i.e.,* DataNode) with the compute resources (*i.e.,* tasks launched by Executors) as shown in Figure 5.1(b). For example, Spark relies on the HDFS [79] to manage the flow of data. HDFS is composed of a master NameNode and many slave DataNodes. Google's MapReduce has a similar reliance on the Colossus, the latest version of Google file system. Such co-localization of DataNodes and Executors realizes a data-centric computing model to minimize data movement between computation tasks and the storage system.

Compared to other MapReduce implementations such as Hadoop [3], Spark provides two key features. First, Spark leverages the distributed memory from all slave nodes to store most intermediate data during job execution and the final execution results at job completion. By doing so, it avoids the file system, retaining most data resident in distributed memory across phases in the same job and/or different jobs. Such *memory-resident* feature benefits many applications such as *machine learning* or *iterative algorithms* that require extensive reuse of results among multiple MapReduce jobs. Second, Spark introduces *resilient distributed datasets* (RDDs) to facilitate the programming of parallel applications. Each RDD represents a collection of data partitions that spread across the cluster. A rich set of operations are provided to manipulate RDDs (*e.g.*, *map*,

flatMap, *groupBy*, and *reduce*, *etc*). Overall, those operations can be categorized into two types, which are *transformation* and *action*, respectively.

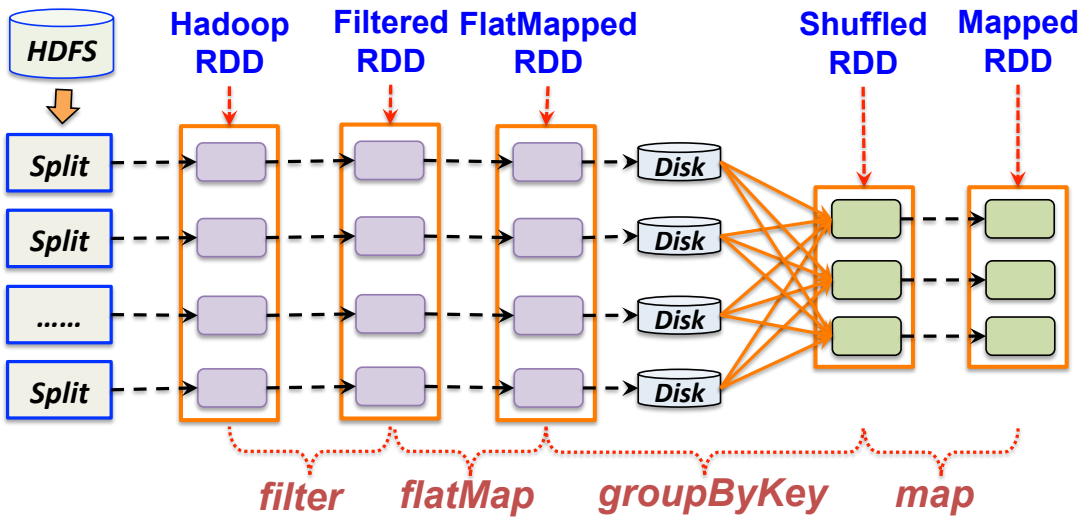


Figure 1.3: MapReduce processing pipeline via using RDDs.

A transformation converts a source RDD to a destination RDD by applying *User-Defined Functions* (UDF) to each partition contained in the former. Figure 1.3 illustrates an example of a Spark MapReduce job, in which an HDFS file is transformed to the final *MappedRDD* through four transformations: *filter*, *flatMap*, *groupByKey* and *map*. When Spark is deployed on a cluster featuring compute-centric paradigm, *HadoopRDD* can be replaced by system dependent RDD, such as *LustreRDD*, to retrieve input from HPC parallel file system.

Spark's actions include *reduce*, *count*, *collect*, *etc*. An action triggers Spark to construct an execution plan represented internally as a *directed acyclic graph* (DAG) that consists of multiple stages. Each stage includes many transformations that can be pipelined. Stages are connected through the shuffle operations for intermediate data shuffling. An implicit stage is embedded into the DAG for every shuffle operation. For example, *filter* and *flatMap* in Figure 1.3 are grouped into a same stage, while the *groupByKey* is in an independent stage. Sparks launches stages within the DAG in a serialized manner.

The shuffling of intermediate data is a major performance bottleneck of MapReduce implementations, including Spark. However, such shuffle operation widely exists in many critical operations, such as *join*, *reduceByKey*, and *groupBy*, *etc*. To avoid substantial overhead and provide reliable job execution, Spark materializes partitions onto the local file system. When a shuffle operation is encountered, Spark will undertake two phases for moving intermediate data: storing and shuffling. In the storing phase, Spark schedules a round of *ShuffleMapTasks* to flush in-memory output from the previous stage to the file system. Then in the shuffling phase, a *ShuffledRDD* is introduced to transfer the intermediate data across the network.

1.2 Overview of Technical Challenges

This dissertation aims to comprehensively optimize the design of MapReduce frameworks from three aspects: (1) exploiting high-performance I/O layer for accelerating intermediate data movement, (2) enhance task management for provisioning quality-of-service to concurrent workloads in multi-tenant MapReduce environments, (3) optimizing the adaptability of MapReduce on HPC platforms. By addressing these three challenges, our optimization can effectively enhance the performance from three dimensions, including job execution time, fairness, and cluster utilization.

1.2.1 How to Optimize Intermediate Data Movement

MapReduce systems have been highly optimized by many designs [102, 67, 80] to reduce the amount of network traffic when reading input data for MapTasks and writing output from ReduceTasks. For instance, delay scheduling [102] helps improve the data locality and reduce data movement in the network. According to their experiment report on many large clusters, up to 98% MapTasks can be launched with inputs on local disks. In addition, ReduceTasks usually generate and store the final outputs to the disks local to themselves in the distributed file systems.

However, intermediate data shuffling causes a large volume of network traffic and remains as a critical bottleneck of MapReduce systems. Every ReduceTask fetches data segments from all map outputs, resulting in a network traffic pattern from all MapTasks to all ReduceTasks, which

grows in the order of $O(N^2)$, assuming that MapTasks and ReduceTasks are both a factor of N total tasks. As reported by [71] from Yahoo!, the intermediate data shuffling from 5% of large jobs can consume more than 98% network bandwidth in a production cluster, and worse yet, MapReduce performance degrades non-linearly with the increase of intermediate data sizes. Also, as reported by [32], network bandwidth oversubscription can quickly saturate the network links of those machines that participate in the reduce phase, This intermediate data shuffling essentially becomes the dominant source of network traffic and performance bottleneck in MapReduce.

Furthermore, at the receiving end of the data shuffling, many well-recognized MapReduce implementations, including [3, 13], resort to disks to absorb fetched intermediate data while Spark leaves merging option to users. However, due to slow performance of disk devices, intermediate data merging is substantially detrimental to the performance of ReduceTasks as pointed out by many studies [57, 90], leading to severely degraded job performance. Slow merging process can significantly delay the progress of ReduceTask to enter into the reduce phase and also incur large amount of small random read during the computation stage. Although leveraging memory can relieve the disk bottleneck issue, how to efficiently use limited storage space is challenging.

1.2.2 How to Achieve both Efficiency and Fairness

Existing MapReduce clusters are no longer solely used for single user single job environment. Instead, they are being shared among many users and running a mix of diverse types of concurrent workloads, including batch jobs and interactive queries in parallel. Such sharing is motivated by many desirable features, such as statistical multiplexing [86] and data consolidation.

To cope with such trend, many scheduling policies [102, 46, 92, 87] have been proposed to optimize multiple metrics, sometimes conflicting, simultaneously for the concurrent MapReduce workloads so that they can deliver good quality-of-services. However, little work has been carried out to improve both job execution time and fairness among multiple jobs these two most critical performance metrics together due to the complexity. Improving the performance of a single job generally requires provisioning more resources to accelerate the processing of its tasks. While, on

the contrary, maintaining the fairness deprives jobs of available resources, leading to degraded job execution times. Thus, it is challenging to balance between efficiency and fairness. Hadoop Fair Scheduler [10] introduced by Facebook and Hadoop Capacity Schedulers [9] initiated by Yahoo! are two notable effort to improve both these metrics, however, our studies show that they are still far away from delivering the optimal performance.

Many issues are hindering MapReduce schedulers from gaining both efficiency and fairness. Firstly, existing solutions assume tasks are short, thereby only assigning tasks when the resources are released by previous tasks. However, the oblivity of distinct execution patterns between Map and Reduce tasks can cause job monopolization problem, penalizing both efficiency and fairness. Secondly, current fair scheduling policies rigidly balance resources among jobs in a weighted fair sharing manner without considering the progress of each individual job and leveraging the lessons from previous scheduling research [40, 83]. However, when certain small jobs are very close to complete, prioritizing them can effectively improve the efficiency with negligible fairness violation. Therefore, current solutions lack the capability to opportunistically enhance the efficiency. Thirdly, once tasks are assigned, they occupy the resource until completion or failure, and existing schedulers do not take account of the utilization of the taken resources, thus incurring resource underutilization when tasks are long running and exhibit intermittent I/O execution patterns.

1.2.3 How to Optimize MapReduce for HPC Platforms

Although MapReduce was initially introduced for commodity clusters, it has also gained broad attention from scientists at the leadership computing facilities as a promising solution to analyze gigantic simulation results. Thus, there is a growing demand for adapting MapReduce systems for High-Performance Computing (HPC) platforms widely deployed in the national laboratories over the past few years.

However, the performance characteristics of MapReduce running on HPC environments remain unknown with little documented literature research [35, 75]. More importantly, the design paradigm of MapReduce frameworks diverges from that of HPC systems substantially. In sum,

there are two key distinctions between MapReduce frameworks and HPC systems. First, there is a key difference on the deployment of compute and storage resources. Second, there is a major difference in terms of the impact of task scheduling and data locality.

Given these distinctions, it is challenging to adapt MapReduce frameworks for HPC platforms since simply deploying MapReduce on the systems is unable to fully exploit the performance advantages of HPC clusters. Thus, in order to enhance the adaptability, it is imperative to characterize the performance of MapReduce on HPC platforms. In particular, it is critical to comprehend the impact of HPC storage systems on the input retrieving, intermediate data storing and shuffling, as well as output sinking. Besides, different from traditional MapReduce clusters that are usually equipped with commodity Gigabit Ethernets, HPC systems, in generally, contain high-performance interconnects, such as InfiniBand, which indicate different performance impact on the locality-oriented scheduling. Therefore, it is also important to obtain understanding how this distinction can affect conventional MapReduce task management policies. Furthermore, HPC systems are evolving to be suitable for MapReduce model. One example is that many HPC clusters are embracing a hierarchy of different types of storage devices on traditional memory-only compute nodes. Thus, it is also of critical importance to study the performance impact of such trend on the MapReduce frameworks.

1.3 Research Contributions

In this dissertation, we have thoroughly investigated the performance of existing MapReduce frameworks on different platforms, revealed their key limitations and bottlenecks, and compared the performance of our solutions with the state-of-the-art. In particular, this dissertation has made following three contributions.

1.3.1 Network Levitated Merge Algorithm

We optimize the intermediate data processing pipeline via introducing a novel *Network Levitated Merge algorithm* [90, 99], along with a *Hadoop Acceleration framework* [7]. The new merge

algorithm overcomes three major inefficiencies in ReduceTask. These include (1) a serialization barrier that delays the reduce phase, (2) repetitive merges and disk access that incur excessive I/O, and (3) the lack of capability to leverage high-performance network protocols. Its feats include merging all the intermediate data entirely in memory and enabling a full pipeline of shuffle, merge, and reduce three phases, thereby efficiently eliminating the disk bottleneck existing in the ReduceTasks and accelerating the progress of entire job. Our experimental results show that the new merge algorithm and Hadoop accelerating framework together double the data processing throughput of Hadoop, reduce CPU utilization, meanwhile maintaining comparable scalability as original Hadoop MapReduce. In addition, Hadoop Acceleration framework has laid down a solid foundation for future research on improving the I/O performance of MapReduce, the work built on top of it has been published in [90, 89, 96, 59, 68, 100, 99, 94]

1.3.2 Preemptive ReduceTask based Fast Completion Scheduler

MapReduce adopts a two-phase (map and reduce) scheme to schedule tasks among data-intensive applications. However, under this scheme, Hadoop schedulers, such Hadoop Fair Scheduler [10] and Hadoop Capacity Schedulers [9], do not work effectively for both phases. We reveal that there exists a serious fairness issue among jobs of different sizes, leading to prolonged execution for small jobs, which are starving for reduce slots held by large jobs. Therefore,they are insufficient to deliver the optimal quality-of-service under multi-tenant MapReduce clusters. To solve this fairness issue and ensure fast completion for all jobs, we have designed the *Preemptive ReduceTask* mechanism and the *Fair Completion scheduler* [91, 82]. Preemptive ReduceTask is a mechanism that corrects the monopolizing behavior of long running reduce tasks from large jobs. Based on the Preemptive ReduceTask, the Fair Completion Scheduler dynamically balances the execution of different jobs for fair and fast completion. Experimental results with a diverse collection of concurrent workloads demonstrate that these techniques together speed up the average job

execution by as much as 39.7%, and improve fairness by up to 66.7%. A comprehensive theoretical analysis of the benefits provisioned by preemptive scheduler for MapReduce have also been documented in [82, 83].

1.3.3 Characterizing and Optimizing MapReduce on HPC Platforms

Recently, MapReduce has gained broad attention from scientists at the U.S. leadership computing facilities as a promising solution to process gigantic simulation results. However, conventional high-end computing systems are constructed based on the compute-centric paradigm while MapReduce frameworks prefer a data-centric paradigm. In this dissertation, we have characterized the performance impact of key differences between compute- and data-centric paradigms and then provided optimizations to enable a dual-purpose HPC system that can efficiently support conventional HPC applications and new data analytics applications, thus, enhancing the adaptability of MapReduce frameworks for the HPC environments to increase cluster utilization. Using *Spark* and the Hyperion system at Lawrence Livermore National Laboratory, we have thoroughly examined the impact of different storage architectures, locality-oriented scheduling to the memory-resident MapReduce jobs. Based on our characterization and findings of the performance behaviors, we have introduced two optimization techniques, namely *Enhanced Load Balancer* and *Congestion-Aware Task Dispatching*, to improve the performance of Spark applications via better utilizing the resources provisioned by HPC clusters.

1.3.4 Publication Contributions

During my Ph.D. study, my research has contributed to the following pulications (listed in the chronological order):

1. Yuan Tian, Scott Klasky, Jay Lofstead, Ray Grout, Norbert Podhorszki, Qing Liu, Yandong Wang, Weikuan Yu. EDO: Improving Read Performance for Scientific Applications Through Elastic Data Organization. IEEE Cluster Computing 2011 [85].

2. Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, Dhiraj Sehgal. Hadoop Acceleration Through Network Levitated Merge. IEEE/ACM the International Conference for High Performance Computing, Networking, Storage and Analysis 2011 [90].

3. Xinyu Que, Yandong Wang, Cong Xu, Weikuan Yu. Hierarchical Merge for Scalable MapReduce. Workshop on Management of Big Data Systems, in conjunction with ICAC 2012 [68].

4. Weikuan Yu, Yandong Wang, Xinyu Que. Design and Evaluation of Network Levitated Merge for Hadoop Acceleration. IEEE Transactions on Parallel and Distributed System 2012 [99].

5. Yandong Wang, Yizheng Jiao, Cong Xu, Xiaobing Li, Teng Wang, Xinyu Que, Cristi Cira, Bin Wang, Zhuo Liu, Bliss Bailey, Weikuan Yu. Assessing the Performance Impact of High-Speed Interconnects on MapReduce Programs. Third Workshop on Big Data Benchmarking 2013 (Invited Book Chapter) [89].

6. Zhuo Liu, Bin Wang, Teng Wang, Yuan Tian, Cong Xu, Yandong Wang, Weikuan Yu. Profiling and Improving I/O Performance of a Large-Scale Climate Scientific Application. the 22nd International Conference on Computer Communication and Networks 2013 [61].

7. Cong Xu, Manjunath Venkata, Richard Graham, Yandong Wang, Zhuo Liu, Weikuan Yu. SLOAVx: Scalable LOgarithmic AlltoallV Algorithm for Hierarchical Multicore Systems. the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing 2013 [95].

8. Yandong Wang, Cong Xu, Xiaobing Li, Weikuan Yu. JVM-bypassing Shuffling for Hadoop Acceleration. the 27th IEEE International Parallel and Distributed Processing Symposium 2013 [96].

9. Yandong Wang, Jian Tan, Weikuan Yu, Li Zhang, Xiaoqiao Meng. Preemptive ReduceTask Scheduling for Fair and Fast Job Completion. USENIX Symposium on International Conference on Autonomic Computing 2013 [91].

10. Xiaobing Li, Yandong Wang, Yizheng Jiao, Cong Xu, Weikuan Yu. CooMR: Cross-Task Coordination for Efficient Data Management in MapReduce Programs. IEEE/ACM The International Conference for High Performance Computing Networking, Storage and Analysis 2013 [59].

11. Weikuan Yu, Yandong Wang, Xinyu Que, Cong Xu. Virtual Shuffling for Efficient Data Movement in MapReduce. IEEE Transactions on Computers 2014 [82].

12. Yandong Wang, Robin Goldstone, Weikuan Yu, Teng Wang. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. the 28th IEEE International Parallel and Distributed Processing Symposium 2014 [97].

13. Jian Tan, Yandong Wang, Weikuan Yu, Li Zhang. Non-work-conserving Effects in MapReduce: Diffusion Limit and Criticality. ACM Sigmetrics 2014 [83].

14. Yandong Wang, Jian Tan, Weikuan Yu, Li Zhang. Achieving Fair and Fast Completion Through Work-Conserving ReduceTask Preemption (under review).

15. Zhuo Liu, Fang Zhou, Yandong Wang, Xiaoning Ding, Weikuan Yu. Two-Level Scheduling of Analytic Queries Through Cross-Layer Semantics Percolation (under review).

16. Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, Weikuan Yu. BurstMem: A High-Performance Burst Buffer System for Scientific Applications (under review).

1.4 Dissertation Overview

In the rest of this dissertation, we detail the problems that prevent MapReduce frameworks from achieving the optimal performance, and then provide detailed description of our techniques. Every chapter focuses on presenting one solution, along with a thorough examination of the performance comparison between our solution and the state-of-the-art techniques.

In Chapter 2, we comprehensively examine the performance and design issues in contemporary MapReduce frameworks to motivate our innovations.

In Chapter 3, we introduce Hadoop Acceleration framework and the novel Network Levitated Merge algorithm that conquer three issues, including serialization barrier, repetitive disk-based merge, and lack of the capability to leverage fast network protocols. Our performance evaluation demonstrates that our solutions can efficiently improve the performance of MapReduce from the perspective of job execution time without sacrificing the other desirable performance metrics.

In Chapter 4, we introduce a lightweight work-conserving Preemptive ReduceTask that provides flexible task execution control and Fast Completion scheduler that leverages the advantage of Preemptive ReduceTasks to provide both efficiency and fairness to jobs of different sizes. Our experiments with different types of concurrent workloads demonstrate the effectiveness of our solutions to improve these two conflicting performance metrics.

In Chapter 5, we thoroughly investigate the performance of Spark on the HPC platforms. As a result of the performance investigation, we reveal many unknown characteristics of MapReduce frameworks when they are deployed on HPC environments. In particular, we have studied the performance implication of different storage architectures, consistency guarantee from parallel file systems, and locality-oriented scheduling policies. Based on our finding, we introduce several optimization techniques to enhance the adaptability of MapReduce frameworks for the HPC platforms to increase cluster utilization.

Eventually, We conclude this dissertation and present opportunities for future work in Chapter 6 and Chapter 7.

Chapter 2

Problem Statement

This chapter discusses the detailed issues that prevent contemporary MapReduce frameworks from overcoming the aforementioned challenges in Chapter 1.2. First, it reveals the issues in Hadoop intermediate data processing pipeline that hinder Hadoop from efficiently moving the intermediate data. Then it explains the design inefficiencies in Hadoop schedulers that result in job starvation and resource underutilization, leading to degraded job execution time and unfairness among concurrent workloads. Finally, it discusses the issues involved for adapting MapReduce frameworks for HPC platforms.

2.1 Issues Preventing Efficient Data Movement

Hadoop's MapReduce implementation enables a convenient and easy-to-use data processing framework. However, with an extensive examination of Hadoop MapReduce framework, particularly its ReduceTasks, we reveal that the original architecture faces a number of challenging issues to exploit the best performance from the underlying system: (1) To ensure the correctness of MapReduce, no ReduceTasks can start reducing data until all intermediate data have been fetched to local and merged together. This results in a serialization barrier that significantly delays the reduce operation of ReduceTasks. (2) More importantly, the current merge algorithm in Hadoop merges intermediate data segments from MapTasks when the number of available segments (including those that are already merged) goes over a threshold. These segments are spilled to local disk storage when their total size is bigger than the available memory. While, this algorithm can cause data segments to be merged repetitively, and therefore multiple rounds of disk accesses of the same data, leading to excessive disk I/O. (3) Current Hadoop architecture can only support TCP/IP protocol to transfer the intermediate data without the capability to leverage the high-performance

network protocols commonly used in High-Performance Computing community. In this following, we discuss these three issues in detail.

2.1.1 A Serialization Barrier in Data Processing

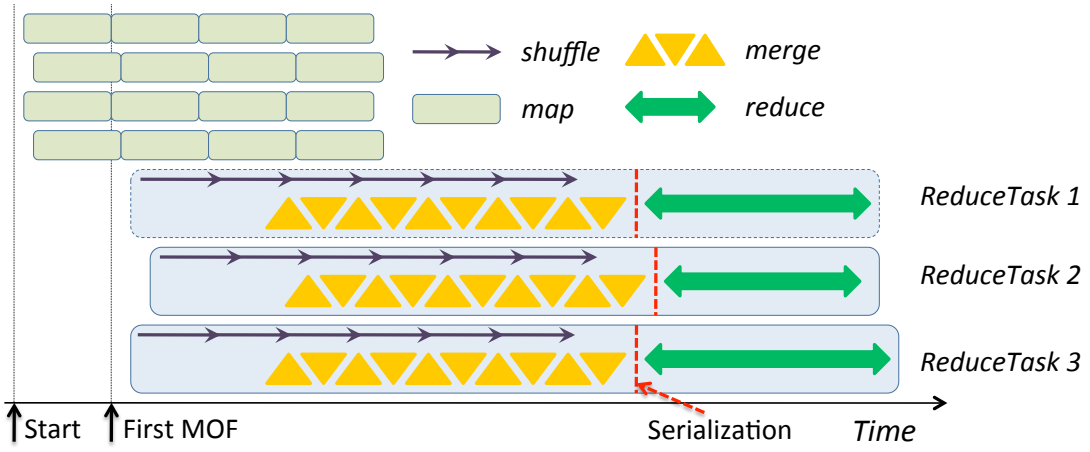


Figure 2.1: Serialization between Shuffle/Merge and Reduce Phases

Hadoop strives to pipeline the processing of large datasets. It is indeed able to do so, particularly for map and shuffle/merge phases. As shown in Figure 2.1, after a brief initialization period, a pool of concurrent MapTasks starts the map function on the first set of data splits. As soon as the Map Output Files (MOFs) are generated from these splits, a pool of ReduceTasks starts to fetch partitions from these MOFs. Within each ReduceTask, there are multiple merging threads running in the background, when the number of segments is larger than a threshold, or when their total data size is more than a memory threshold, the smallest segments are merged.

To guarantee correctness of the MapReduce programming model, it is necessary to ensure that the reduce phase does not start until the map phase is done for all data splits. However, the pipeline as shown in Figure 2.1 contains an implicit serialization. At each ReduceTask, not until all its segments are available and merged, will the reduce phase start to process data segments via the reduce function. These essentially enforce a serialization between the shuffle/merge phase and the reduce phase. When there are many segments to process (which is often the case), it takes a

significant amount of time for a ReduceTask to shuffle and merge them. As a result, the reduce phase will be significantly delayed. Our analysis (c.f. Section 3.4.3) has revealed that this can delay the reduce phase by 668 seconds, i.e., more than 39.4% of the total execution time for a Hadoop program that sorts 192GB of data on 24 nodes (c.f. the last row of Table 3.2).

2.1.2 Repetitive Merges and Excessive Disk Access

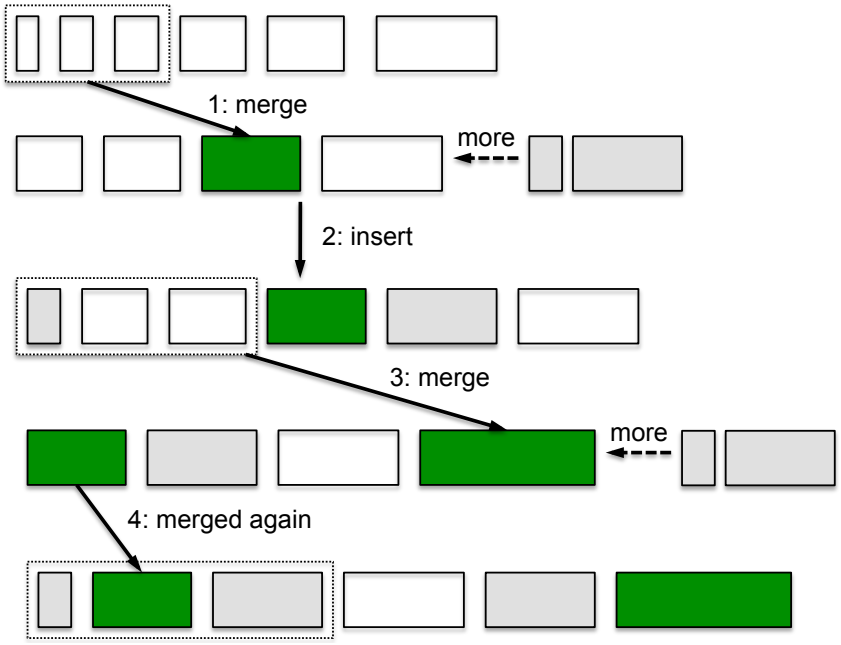


Figure 2.2: Repetitive Merging and Disk Access

As mentioned earlier in the overview, ReduceTasks merge data segments when the number of segments or their total size goes over a threshold. A newly merged segment has to be spilled to local disks due to memory pressure. However, the current merge algorithm in Hadoop often leads to repetitive merges, thus extra disk accesses. Figure 2.2 shows a common sequence of merge operations in Hadoop. For the purpose of illustration, we hereby choose a very small threshold parameter $io.sort.factor = 3$ (parameter is defined as *mapreduce.task.io.sort.factor* in latest YARN MapReduce that suffers from the same repetitive merge issue). A ReduceTask fetches its data segments and arranges them in the order of their size. When the number of data segments reaches six, i.e., twice the threshold, the smallest three segments are merged, shown as Step 1 in Figure 2.2.

Under memory pressure, this will incur disk access. The resulting segment is inserted back into the heap based on its relative size.

When more segments arrive (as shown in Step 2), the threshold will be broken again. It is then necessary to merge another set of segments, shown as Step 3. This again causes additional disk access, let alone the need to read some segments if they have been stored on local disks. Depending on its relative size, a previously merged segment is likely to be grouped into another set and merged again, shown as Step 4. Since the smallest segments are usually selected for merge, chances are rather high for a segment to be merged repetitively. Furthermore, any segment merged from a subset of segments eventually needs to be merged for final results. Altogether, this means repetitive merges and disk access, therefore degraded performance for Hadoop.

Table 2.1: Profile of Excessive I/O During Merging

Total Number of Segments	480
Intermediate data per ReduceTask	5.69GB
Percentage of segments that are merged once	100%
Percentage of segments that are merged Twice	98.1%

To illustrate the excessive I/O caused by the existing merge algorithm in Hadoop, we have conducted an experiment running TeraSort with 120GB input data across 20 nodes. We count the number of partitions that are merged at least once and measure the data size involved in the merging process. Table 2.1 shows the profiling results. On average, each ReduceTask needs to fetch 480 partitions from all the MapTasks and processes up to 6GB intermediate data. Before a ReduceTask starts its reduce phase, we observe that all the partitions are merged at least once from memory to disk and up to 98.1% of partitions are merged twice. With an average intermediate data size of 5.69GB, each ReduceTask has to write such data to and read from the disks several times. Such excessive I/O aggravates the interference among tasks and delay the execution of entire MapReduce programs.

It is tempting to choose a different policy for merge. This can lead to a similar problem of essentially the same nature. The key constraint is that, if any merge happens before a global order

of segments is established, it ought to be re-merged into the final result before the reduce function. Therefore, an alternative merge algorithm is critical for Hadoop to mitigate the impact of repetitive merges and their associated excessive disk access behavior.

2.1.3 Unable to Leverage RDMA Interconnects

Furthermore, Hadoop does not take advantage of high-performance RDMA [72] interconnect technologies such as InfiniBand [44] that have matured in the HPC community. For example, a node in a hierarchical Hadoop cluster is typically equipped with one or more Gigabit Ethernet (GigE) network interface cards and connected to the lowest tier GigE switch. With this configuration, one card can only add an upper bound of 125 MB/sec to the data movement throughput of Hadoop. Given a multi-socket and multi-core server, such capacity has to be shared and very thinly divided amongst all cores. Worse yet, advances in processor technology will soon deliver compute servers with hundreds of cores to the mass market. Furthermore, RDMA supports high bandwidth data movement with very little CPU involvement. Simply replacing the network hardware with the latest interconnect technologies such as InfiniBand and 10 Gigabit Ethernet, and continuing to run Hadoop on TCP/IP will not enable Hadoop to leverage the strengths of RDMA. Thus, the lack of support for RDMA interconnects will become a severe threat for Hadoop to keep up with the advances of other computer technologies, particularly when more highly capable processors, storage, and interconnect devices are deployed to various computing and data centers.

2.2 Issues in Hadoop Task Management

To support many users and jobs (large batch jobs and small interactive queries), Hadoop MapReduce adopts a two-phase (map and reduce) scheme to schedule tasks for data-intensive applications. The Hadoop Fair Scheduler (HFS) [10] and Hadoop Capacity Scheduler (HCS) [9] have focused on fairness among MapTasks. These schedulers strive to maximize the use of system capacity and ensure fairness among different jobs. However, they do not work effectively for both phases. What complicates the matter is the distinct execution behaviors between MapTasks and

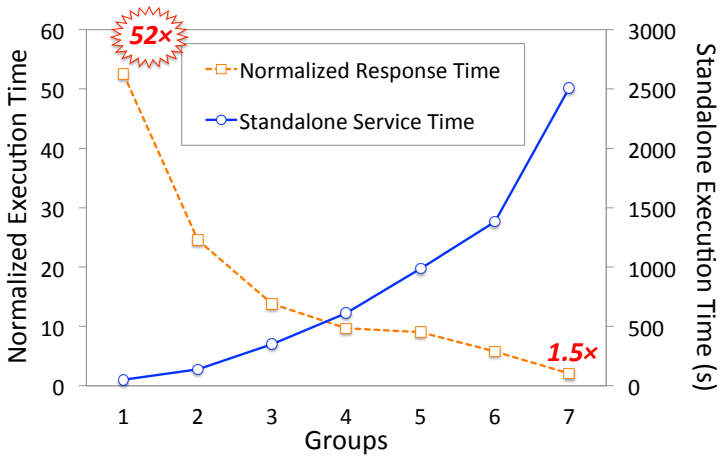


Figure 2.3: Unfair Execution among Different Size Jobs

ReduceTasks. Unlike MapTasks which are generally very short-lived and launched one group after the other to process data splits, ReduceTasks have a different execution pattern. As shown by the Facebook trace [102], the execution time of average ReduceTask is longer than that of MapTasks by one order of magnitude. In addition, once a ReduceTask is launched, it occupies the reduce slot until completion or failure.

We have examined the performance of Hadoop schedulers using a synthetic workload of jobs submitted to a shared MapReduce cluster. Jobs are divided into 7 groups based on their increasing data sizes; jobs in the same group are identical. They arrive according to a Poisson random process. Figure 2.3 shows the comparison of the *normalized execution time*, which is defined as the ratio between a job's actual execution time and its stand-alone execution time (the time when a job is running in the system alone). As shown in the figure, the stand-alone execution time of jobs in each group increases in proportion to their input data size. However, the completion of these jobs varies dramatically with HFS. Jobs in the smaller groups have much worse normalized execution times, suggesting that they must wait very long (as much as 52$\times$ longer than the stand-alone execution time). Worse performance results have been observed with HCS, thus we omit its performance for succinctness. Such scheduling behavior contradicts users' intuitive expectation that smaller jobs should be completed faster and turned around more quickly, indicating severe unfairness issue in existing Hadoop MapReduce schedulers.

2.2.1 Unfair Reduce Slot Allocation

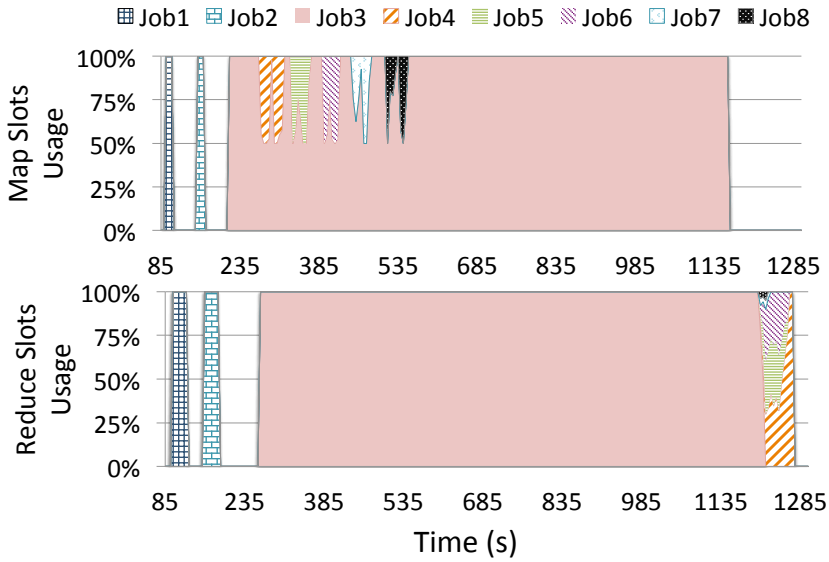


Figure 2.4: Run-Time Allocation Profile of Map and Reduce Slots.

To more closely examine the unfairness issue between different jobs, we conduct another experiment on a cluster of 20 nodes. 40 map slots are created on 10 nodes, and 20 reduce slots on another 10 nodes. 8 jobs are sequentially submitted into the cluster every 60 seconds. Job 3 is a large job that requires 20 ReduceTasks. Figure 2.4 shows the usage of map and reduce slots by 8 jobs. Map slots are efficiently shared among jobs over time as jobs arrive and leave, but reduce slots are all occupied by Job 3. As a result, Jobs 4-8 cannot get a share until Job 3 completes, even if they have successfully finished all their MapTasks. On average, Jobs 4-8 are severely slowed down by 1486%, compared to their stand-alone execution times. This reveals that Hadoop Fair Scheduler is not able to achieve fair normalized execution times for all jobs. A similar behavior has also been reported by an IBM study [81]. Note that there exists a dramatic variance among the normalized execution time for different jobs in the same pool and in different tests (c.f. Figure 2.3 and Figure 2.4).

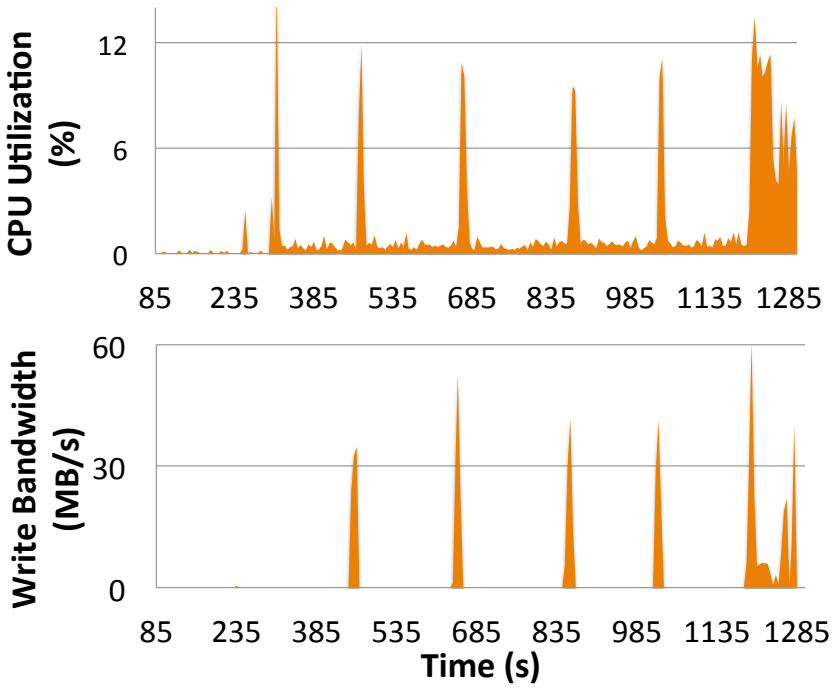


Figure 2.5: Inefficient Usage of Reduce Slot

2.2.2 Resource Underutilization within ReduceTasks

Existing schedulers are also oblivious of the resource underutilization issue incurred by long running ReduceTasks. When the generation rate of intermediate data is low, even when long running ReduceTasks are occupying the slots, they do not efficiently utilize the resources, and ReduceTasks periodically enter into the idle state, causing severe resource underutilization. In this section, we demonstrate such problem.

During the shuffle phase, ReduceTasks only fetch intermediate data from remote MapTasks when there are available map outputs. When intermediate data is unavailable, a ReduceTask enters into the idle status. But it still occupies the slot, blocking other jobs from acquiring the resource and degrading the overall system efficiency. Figure 2.5 presents the average CPU utilization and disk write bandwidth on the machines that host Job 3 in the previous experiment. We can see that, between 235th second and 1135th second (the execution period of Job 3's ReduceTasks), CPUs and disks frequently become idle and are only periodically activated, even though another 5 jobs are still waiting in the queue. In addition, we also observe that network is highly underutilized in

this scenario. On average, during 87.6% of Job 3's ReduceTask execution time, CPUs and disks are idle and waiting for the intermediate data. This problem can be more exacerbated when a large job is competing for map slots with other jobs. This competition for map slots prolongs the execution of the large job and slowers its generation rate of intermediate data, hence further delays the release of reduce slots, stretching all other jobs' wait time.

2.2.3 Insufficiency in Existing Solutions

The monopolizing behavior of ReduceTasks has been documented earlier as a reason to cause small jobs starve for reduce slots [92, 81, 101]. Hadoop provides a slowstart configuration option that can delay the launch of ReduceTasks and mitigate this situation, but at the cost of slowing down the shuffle phase, thus it can significantly prolong the execution times of small jobs. Zaharia *et al.* [101] proposed a copy-compute splitting mechanism, but it does not fully resolve this issue. Tan *et al.* [81] proposed a coupling scheduler to launch reducers gradually by coupling the progresses of map and reduce tasks in the same job. With this scheduler, a large job can spare reduce slots for other jobs when its map phase has not progressed much. But when a large job finishes its map phase, it still takes all available reduce slots and causes the starvation of small jobs. Like the slowstart option, the coupling scheduler delays and mitigates the monopolization of reduce slots by large jobs. But it does not solve the monopolization of reduce slots by large jobs, instead let it progressively happen. In view of these issues, existing solutions are insufficient to address unfairness and inefficiency issue.

2.3 Issues in Adapting MapReduce for HPC Platforms

Many organizations have been embracing MapReduce and deploying its different implementations to meet their needs of massive computation and analysis of enormous datasets, thereby mining critical knowledge for their business.

In this modern rush for gold from data, different organizations are facing very different considerations when it comes to a decision on their data analytics systems. With the prevalence of cloud

platforms and commercial computing services, many customers can leave that decision to their system providers. But the system providers really have to juggle between two choices: should they construct from scratch dedicated systems for data analytics, or should they evolve their systems to meet the demands of data analytics applications while continuing to support existing applications and customers? The latter is a particularly perplexing situation faced by the users and administrators at the leadership computing facilities who have been relying on traditional HPC systems for their scientific applications. Accompanying this dilemma is that there is a hidden paradigm shift along with the emergent focus on big data. For the first few decades of computer history, computing power has been a scarce resource. Thus conventional systems are constructed based on a compute-centric paradigm while the grand objective is to aggregate as much computing power as possible in terms of the number of floating-point operations per second. The need to analyze big data has actually pushed the transition of computer systems into a data-centric paradigm for which the grand objective is to attain the fastest analytics power in terms of the number of bytes and records processed per second. In the following, we further describe the paradigm distinctions between HPC systems and MapReduce model.

2.3.1 Conflict between Design Paradigms

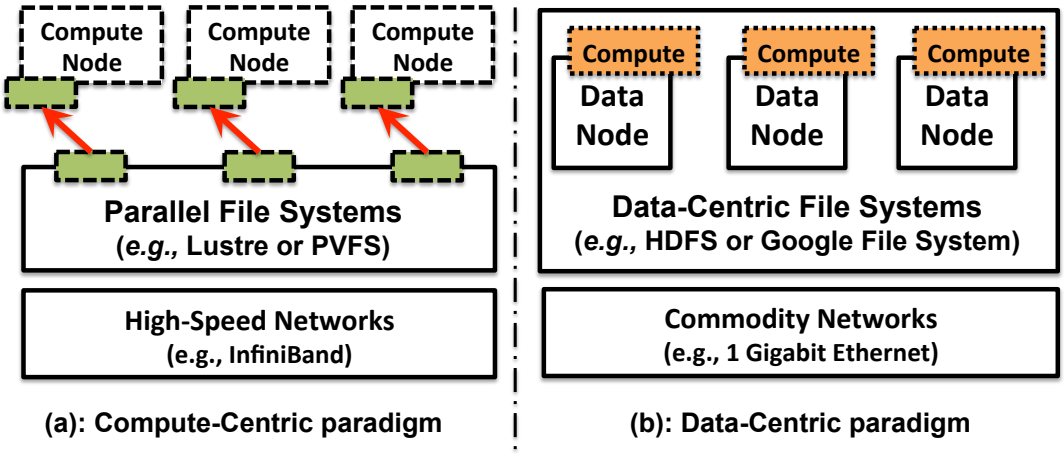


Figure 2.6: Data-centric and compute-centric paradigms.

Figure 2.6 shows a comparison between compute- and data-centric paradigms. There are two key distinctions between these paradigms. First, there is a key difference on the placement of compute and storage resources. The conventional compute-centric paradigm has separated compute and storage resources in the form of a computer cluster and a parallel file system that are connected via high speed networks. In contrast, the data-centric paradigm provides co-located compute and storage resources on the same node. Second, there is a key difference in terms of the impact of task scheduling and data placement. In the compute-centric paradigm, tasks on compute nodes are, in general, equally distant from the backend storage system. HPC systems are Typical manifestations of this paradigm. In the data-centric paradigm, tasks have strong affinity to the nodes containing their datasets. Because of these distinctions, on compute-centric paradigm, applications sharing the same data often involve repetitive data movement between the computing resource and the storage backend. In contrast, the data-centric paradigm provides co-located compute and storage resources on the same node to facilitate locality-oriented task scheduling. By scheduling computing tasks to where data resides, data movement can be minimized for applications sharing the data, benefiting the MapReduce frameworks substantially on commodity clusters.

These distinctions between compute- and data-centric paradigms have significant performance implications to different types of application workloads. For system providers who are eager to support more MapReduce-based analytics applications on HPC platforms, it is imperative to characterize the performance of key architectural components in these two different paradigms. Particularly, how does the configuration of storage resources such as parallel file systems affect job scalability and throughput? What is the impact of data placement and task scheduling? And how to reconcile and converge the architectural differences between the two paradigms so that one system can be configured and tuned for productive sharing by both conventional HPC applications and the emergent MapReduce-based analytics applications.

2.4 Summary

In summary, this chapter has revealed the issues that lead to poor intermediate data processing pipeline, unfairness among concurrent workloads, and conflict between the design paradigms of MapReduce and HPC systems. Therefore, this dissertation is devoted to addressing those issues, thus achieving simultaneous multi-dimensional performance improvement for MapReduce frameworks. To be more specific, this dissertation seeks optimization solutions to tackle above issues from the following three directions.

- Fast data processing pipeline for MapReduce frameworks
- Efficient job scheduling for provisioning both efficiency and fairness
- Enhancing the adaptability of MapReduce for HPC platforms

The solutions are:

- Hadoop Acceleration framework and Network Levitated Merge algorithm
- Preemptive ReduceTask based Fast Completion Scheduler
- Characterizing and Optimizing MapReduce for HPC Platforms

Chapter 3

Network Levitated Merge Algorithm

3.1 Introduction

As described in Chapter 1.1.2, Hadoop [3] is an open-source implementation of MapReduce, currently maintained by the Apache Foundation, and endorsed by many leading IT companies. Hadoop implements MapReduce framework with two major categories of components: a JobTracker and many TaskTrackers. The JobTracker commands TaskTrackers to process data in parallel through two map and reduce functions. In this process, the JobTracker is in charge of scheduling MapTasks and ReduceTasks to TaskTrackers, meanwhile monitoring their progress, collects runtime execution statistics, and handles possible faults and errors through task re-execution.

Performance and scalability are critical to ensure Hadoop's continuing success to various industry and scientific users. A number of studies have been carried out to improve the performance of Hadoop MapReduce framework. Jiang et al. [48] have tuned the parameters of Hadoop for better performance. Condie et al. [30] have proposed the *MapReduce Online* architecture to open up direct network channels between MapTasks and ReduceTasks and speed up the delivery of data from MapTasks to ReduceTasks. While their work reduces job completion time and improves system utilization, it cannot cope with a gigantic dataset that does not fit in memory, and also complicates the fault tolerance handling of Hadoop tasks. Furthermore, it demands a large number of network channels for data movement. For these reasons, MapReduce Online has to fall back onto the default MapReduce execution mode.

Furthermore, little work has been carried out to examine the relationship of Hadoop MapReduce's three data processing phases, i.e., shuffle, merge, and reduce, and their implication to the efficiency of Hadoop. As illustrated in Chapter 2.1. Current Hadoop intermediate data processing pipeline suffers from three critical performance issues, including (1) a serialization barrier between

shuffle/merge and reduce phases that can severely delays the progress of reduce phase. (2) repetitive merge and excessive disk I/O issue in merge algorithm used by current ReduceTasks. (3) the incapability to leverage high-performance network protocols, such as RDMA.

To address these critical issues for Hadoop MapReduce Framework, we have designed Hadoop-A, an acceleration framework that can take advantage of plugin components for performance enhancement and protocol optimizations. Several enhancements are introduced: (1) a novel Network Levitated Merge algorithm that enables ReduceTasks to perform data merging without repetitive merges and disk access; (2) this new merge algorithm also enables a full pipeline that overlaps the shuffle, merge and reduce phases for ReduceTasks; and (3) besides the TCP/IP protocol in the original Hadoop, an alternative protocol is introduced in Hadoop-A to enable data movement via RDMA (Remote Direct Memory Access). Since ReduceTasks are able to merge data by staying above local disks, we refer to this new algorithm as Network Levitated Merge. We have carried out an extensive set of experiments to evaluate the performance of Network Levitated Merge and Hadoop-A compared to the original Hadoop. Our evaluation demonstrates that the new merge algorithm is able to remove the serialization barrier and effectively overlap data merge and reduce operations for Hadoop ReduceTasks. Overall, Hadoop-A is able to improve the throughput of Hadoop data processing by more than 100%. Its RDMA-based data movement also reduces CPU utilization by more than 36%.

The rest of the chapter is organized as follows. We firstly describe the Hadoop-A acceleration framework in Section 3.2, followed by Section 3.3 that details the network-levitated merge algorithm. Section 3.4 provides experimental results. Section 3.5 provides a review of related work. Finally, we summarize this chapter in Section 3.6.

3.2 Design of Hadoop Acceleration

In view of the issues discussed in Section 2.1, we deem that it is important to design a solution that can accelerate Hadoop's MapReduce framework and overcome existing limitations. We have

designed Hadoop-A, an acceleration framework that allows Hadoop to take advantage of RDMA-capable interconnects, experiment with different plugin merge algorithms, and retain the same user interface. In this section, we will describe the architecture of Hadoop-A, and the exploitation of RDMA for data shuffling.

3.2.1 Software Architecture of Hadoop-A

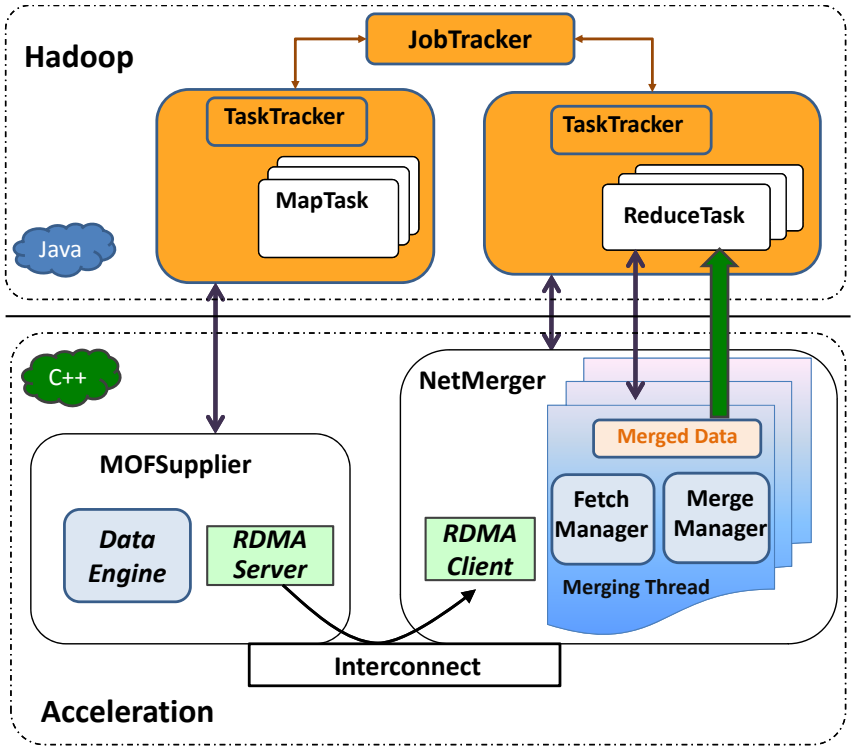


Figure 3.1: Software Architecture of Hadoop-A

Figure 3.1 shows the architecture of Hadoop-A. Two new user-configurable plugin components, *MOFSupplier* and *NetMerger*, are introduced to leverage RDMA-capable interconnects and enable alternative data merge algorithms. Both MOFSupplier and NetMerger are threaded C++ implementations, with all components following the object-oriented principle. The choice of C++ over Java is to avoid the overhead of the Java Virtual Machine (JVM) in data processing and allow flexible choice of new connection mechanisms such as RDMA, which is not yet available in Java.

We briefly describe several features of this acceleration framework without going too much into the technical details of their implementations.

User-Transparent Plugins – A primary requirement of Hadoop-A is to maintain the same programming and control interfaces for users. To this end, we design the MOFSupplier and NetMerger plugins as C++ programs that can be launched by TaskTrackers. A user can choose to enable or disable the acceleration, which is controlled by a parameter in the configuration file. With these run-time plugins, we ensure that Hadoop-A is user-transparent in two ways: (1) no changes are introduced for the scheduling and monitoring interface between TaskTracker and MapTask, and the same for TaskTracker and ReduceTask; and (2) no modification is made to the submission and control interface between a user program and the JobTracker. All MapReduce programs written for Hadoop will continue to function with Hadoop-A.

Multithreaded and Componentized MOFSupplier and Netmerger – MOFSupplier contains an RDMA server that handles fetch requests from ReduceTasks. It also contains a data engine that manages the index and data files for all MOFs that are generated by local MapTasks. Both components are implemented with multiple threads in MOFSupplier. NetMerger is also a multithreaded program. It provides one thread for each Java ReduceTask. It also contains other threads, including an RDMA client that fetches data partitions and a staging thread that uploads data to the Java-side ReduceTask.

Event-Driven Progress and Coordination – To synchronize with Java-side components, we provide event channels between MOFSupplier/NetMerger plugins and Hadoop. These event channels are also used to coordinate activities and monitor progress for internal components of MOFSupplier and NetMerger. All channels are implemented through asynchronous loopback sockets that can wake up threads when there are tasks, and allow them to go back to sleep when tasks are not available. Run-time progress reports and execution statistics are collected and stored as a part of Hadoop logging files. Such logging utilities are capable of monitoring and dissecting the execution of Hadoop jobs. For example, results in Section 3.4.3 are collected by using this feature.

3.2.2 RDMA-Accelerated Data Shuffling

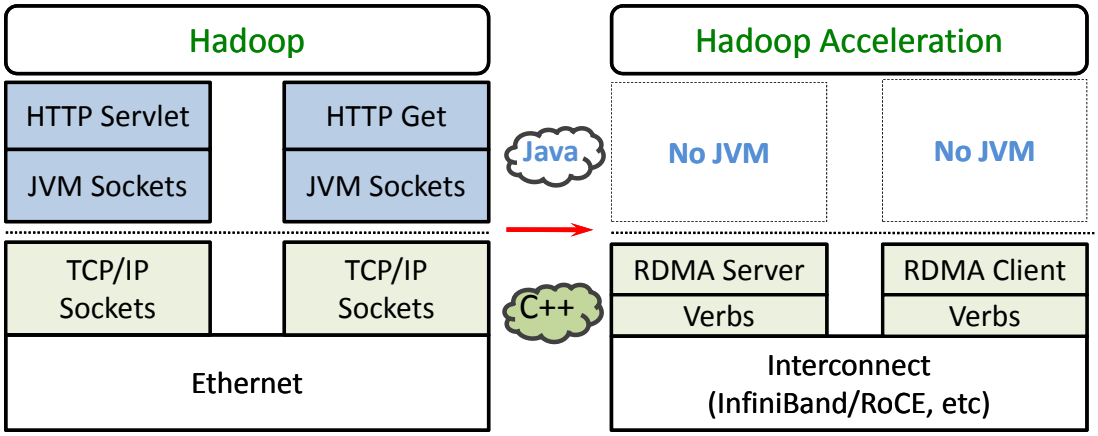


Figure 3.2: RDMA-Accelerated Data Shuffling

As mentioned in Section 2.1.3, Hadoop cannot make use of RDMA interconnects such as InfiniBand. The left half of Figure 3.2 shows the communication stack currently used for Hadoop data shuffling. When notified of the completion of a MOF, Hadoop ReduceTasks invoke a copy thread to fetch its data partition through a Java-based HTTP GET request. On the server side, a Java-based HTTP server is launched by every TaskTracker. A specific HTTP servlet is attached to this server to handle HTTP GET requests and serve data partitions from the MOF files accordingly.

Hadoop-A component architecture allows us to introduce alternative communication protocols for data shuffling in Hadoop. To exploit the benefit of RDMA-capable interconnects, we design our RDMA-based data shuffling protocol completely in the native C++ language, as shown on the right of Figure 3.2. The new protocol directly builds the RDMA-based communication on top of the verbs protocol, and completely avoids the overhead of JVM for Hadoop data shuffling.

RDMA-based shuffling protocol consists of an RDMA server in the MOFSupplier and an RDMA client in the NetMerger. InfiniBand Reliable Connected (RC) service are established on a per-node basis for RDMA clients and servers. The RDMA CM protocol is used for connection establishment. Connections are retained for the lifetime of an RDMA client, and will be torn

down and re-established for a revived client. Once connected, RDMA clients and servers communicate data through pre-registered memory buffers. The data engine in the MOFSupplier always prefetches data segments. It retrieves data from disk when the requested data is not yet available in memory. Such data movement is realized through a direct request and reply protocol. An RDMA client sends a request along with the information of the available memory buffer, and the RDMA server locates the data and writes it to the client buffer via a zero-copy RDMA write operation. More implementation details of the RDMA protocol (and the data engine) are omitted here as they are not the focus of this paper.

3.3 A Network-Levitated Shuffle, Merge and Reduce Pipeline

As discussed in Section 2.1.1 and Section 2.1.2, there exist two critical issues in Hadoop: (1) the serialization barrier between shuffle/merge and reduce phases, and (2) repeated merges and disk access. We first describe a network-levitated merge algorithm that avoids repeated merges, and then detail the construction of a new pipeline to eliminate the serialization.

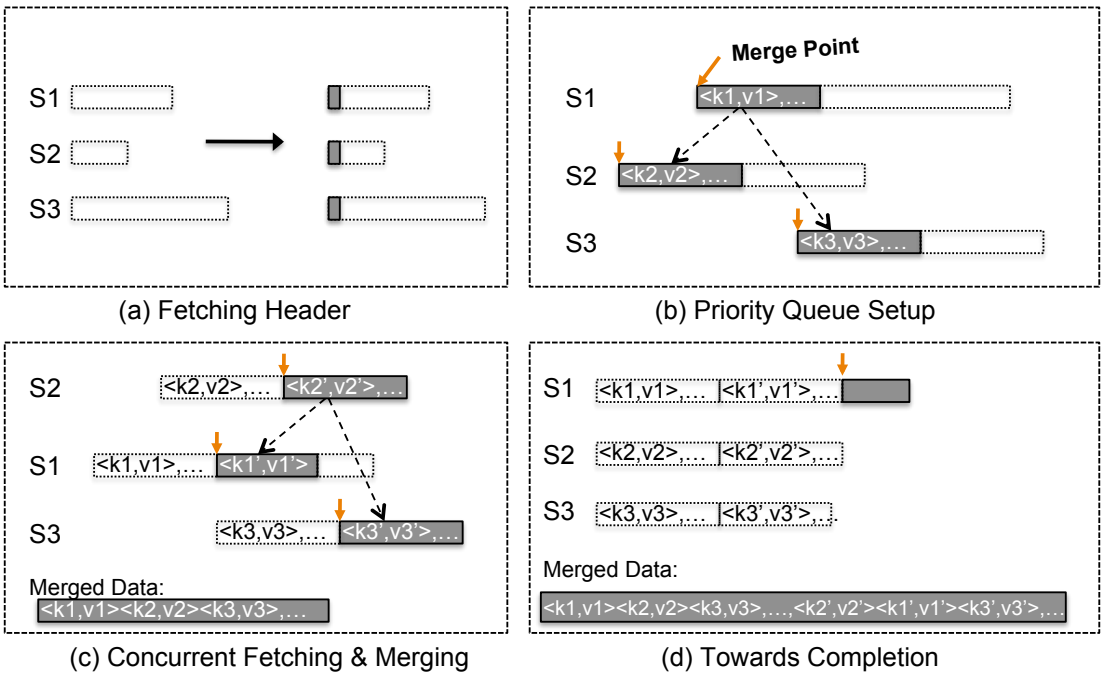


Figure 3.3: A Network-Levitated Merge Algorithm

3.3.1 Network-Levitated Data Merge

Hadoop resorts to repetitive merges because of limited memory compared to the size of data. For each remotely completed MOF, it invokes an *HTTP GET* request to query the partition length, pull the entire data, and store locally in memory or on disk. This incurs many memory loads/stores and/or disk I/O operations. With the performance of RDMA interconnects comes so close to memory, it is now unnecessary, even unwise, to pull data partitions locally before merging. Therefore we design an algorithm that can merge all data partitions exactly once, and at the same time stay *levitated* above local disks.

Figure 3.3 shows our network-levitated merge algorithm. Our algorithm is modified from Hadoop's Priority Queue-based merge sort algorithm. The key idea is to leave data on remote disks until it is time to merge the intended data records.

As shown in Figure 3.3(a), three remote segments S1, S2, and S3 are to be fetched and merged. Instead of fetching them to local disks, our new algorithm only fetches a small header from each segment. Each header is specially constructed to contain partition length, offset, and the first pair of <key,val>. These <key,val> pairs are sufficient to construct a Priority Queue (PQ) to organize these segments. More records after the first <key,val> pair can be fetched as allowed by the available memory. Because it fetches only a small amount of data per segment, this algorithm does not have to store or merge segments onto local disks. Instead of merging segments when the number of segments is over a threshold, we keep building up the PQ until all headers arrive and are integrated. As soon as the PQ has been set up, the merge phase starts. The leading <key,val> pair will be the beginning point of merge operations for individual segments, i.e., the *merge point*. This is shown in Figure 3.3(b).

Our algorithm merges the available <key,val> pairs in the same way as is done in Hadoop. Each segment is a part of a MOF produced by the map function of Hadoop, which means that it is composed of data records ordered by their keys. Thus the root of a complete PQ is the first record among all segments. So, it is safe to store this root record as the first record in the final merged data. When the PQ is updated, the next root will be the first <key,val> among all remaining segments.

It is then stored to the final merged data as well. When the available data records in a segment are depleted, our algorithm can fetch the next set of records to resume the merge operation. In fact, our algorithm always ensures that the fetching of upcoming records happens concurrently with the merging of available records. As shown in Figure 3.3(c), the headers of all three segments are safely merged; more data records are fetched, and the merge points are relocated accordingly.

Concurrent data fetching and merging continue until all records are merged. Note that data records are merged exactly once and stored as part of the merged results. Figure 3.3(d) shows a possible state of the three segments when their merge completes. Naturally, one may ask where the merged data is stored and what happens if it cannot be contained in memory. Since the merge data has the final order for all records, we can safely deliver the available data to the Java-side ReduceTask where it is then consumed by the reduce function. Further details are available below in Section 3.3.2.

3.3.2 Pipelined Shuffle, Merge and Reduce

Besides avoiding repetitive merges, our algorithm removes the serialization barrier between merge and reduce. As described in Section 3.3.1, the merged data has <key,val> records ordered in their final order, and can be delivered to the Java-side ReduceTask as soon as they are available. Thus the reduce phase no longer has to wait until the end of the merge phase.

In view of the possibility to closely couple the shuffle, merge and reduce phases, we design Hadoop-A with a full pipeline, which is shown in Figure 3.4. In this pipeline, MapTasks map data splits as soon as they can. When the first MOF is available, ReduceTasks fetch the headers and build up the PQ. These activities are pipelined. Header fetching and PQ setup are pipelined and overlapped with the map function, but they are very light-weight, compared to shuffle and merge operations. As soon as the last MOF is available, completed PQs are constructed. The full pipeline of shuffle, merge, and reduce then starts. One may notice that there is still a serialization between the availability of the last MOF and the beginning of this pipeline. This is inevitable in order for Hadoop to conform to the correctness of the MapReduce programming model. Simply stated, it is

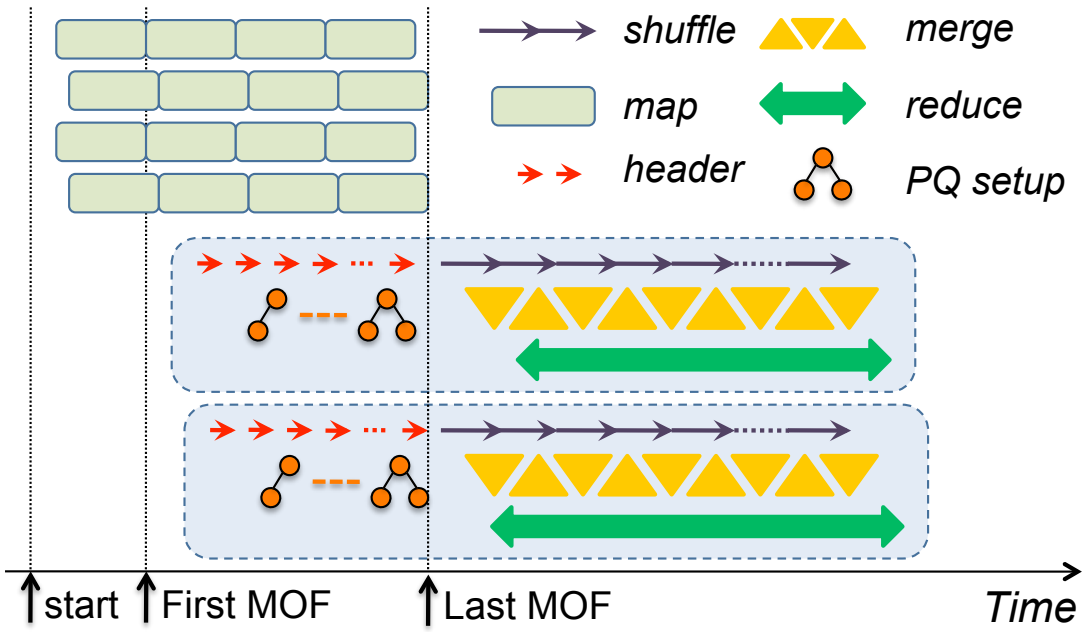


Figure 3.4: Pipelined Shuffle, Merge and Reduce

erroneous to send any data to the reduce function (for final results), while the intermediate result is yet to be produced by the map function.

Therefore our pipeline is able to shuffle, merge and reduce data records as soon as all MOFs are available. This eliminates the previous serialization barrier in Hadoop, and allows intermediate results to be reduced as soon as possible for final results.

3.4 Experimental Results

In this section, we show experimental results from our evaluation of Hadoop-A on InfiniBand, compared to the original Hadoop on Gigabit Ethernet and IPoIB.

3.4.1 Testbed Environment

We conduct our experiments on a cluster of 26 nodes. Each node is equipped with dual-socket quad-core 2.13GHz Intel Xeon processors and 8 GB of DDR2 800 MHz memory, along with 8x PCI-Express Gen 2.0 bus. Four cores on a socket share 4 MB L2 cache. These nodes run Linux

Table 3.1: Comparison of Network Performance

Devices	Bandwidth (MB/sec)	
	Java	C
IB (RDMA)	–	3239.21
IB (IPoIB)	1078.40	1220.39
Gigabit Ethernet	122.31	124.13

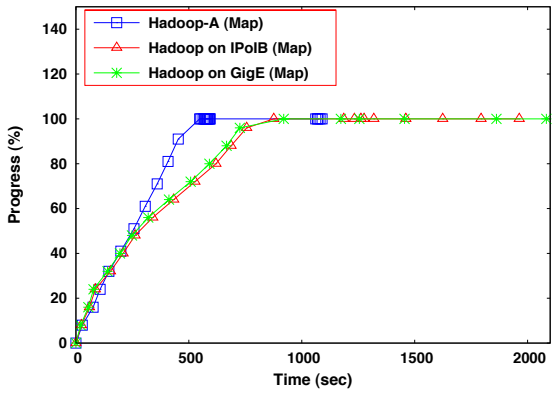
2.6.18-164.el5 kernels. All nodes are equipped with Mellanox ConnectX-2 QDR Host Channel Adaptors and are connected to a 36-port InfiniBand QDR switch. We use the InfiniBand software stack, OFED [14] version 1.5.2, as released by Mellanox. Each node has a 250GB, 7200 RPM, Western Digital SATA hard drive.

The performance of RDMA is measured using the perf_test from OFED [14], that of IPoIB and Gigabit Ethernet (GigE) using the netperf [18] benchmark. For the performance of IPoIB and Gigabit Ethernet in Java, we use a Java-based TTCP benchmark [16].

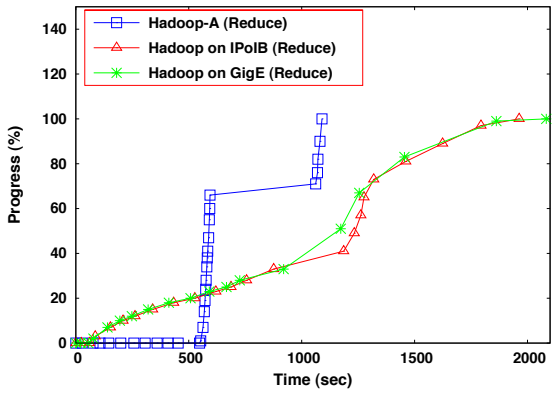
Table 3.1 shows the comparison of peak throughput for three network protocols: RDMA, IPoIB and GigE. RDMA delivers much higher throughput in the C environment, but it is not available in a Java environment. IPoIB can achieve a peak performance of 1078.40 MB/sec and 1220.39 MB/sec, respectively, when running in Java and C. For all our tests, we use the default *connected* mode of IPoIB. GigE can achieve a peak of 122.31 MB/sec and 124.13 MB/sec in Java and C, respectively. Note that compute nodes in our system have relatively slow processors and memory buses compared to the best available in the market. Thus these numbers may differ slightly from vendors' advertised performance numbers. Nonetheless, these network protocols provide a good set to compare the performance of Hadoop and Hadoop-A.

3.4.2 Overall Performance

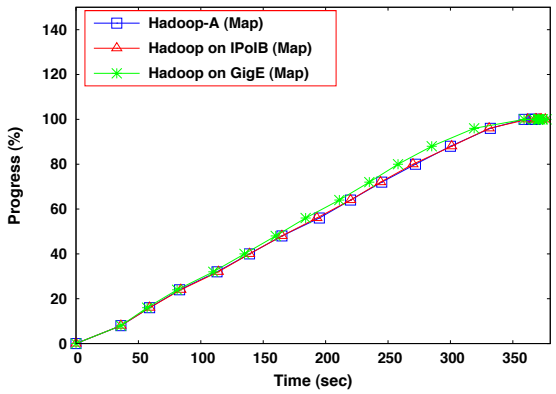
We run Hadoop TeraSort and WordCount programs with different data sizes and different numbers of slave nodes. We choose the data size per split as 256MB. Each slave has 8 MapTasks and 4 ReduceTasks. Figure 3.5 shows the performance comparison between Hadoop-A and Hadoop for TeraSort and WordCount programs. The Y-axis shows the percentage of completion



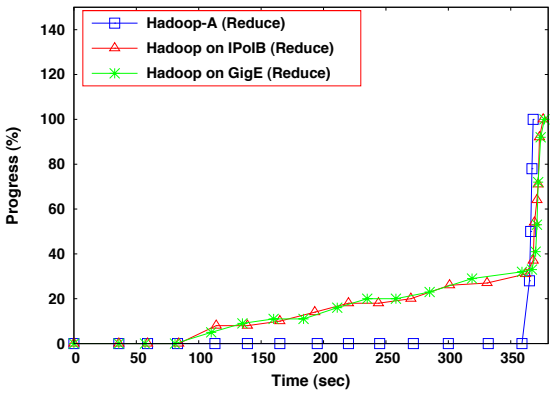
(a) Map Progress of TeraSort



(b) Reduce Progress of TeraSort



(c) Map Progress of WordCount



(d) Reduce Progress of WordCount

Figure 3.5: Progress Diagrams of TeraSort and WordCount Benchmarks

for Map and Reduce Tasks. The X-axis shows the progress of time during execution. As shown by (a) and (b), Hadoop-A speeds up the total execution time significantly for the TeraSort program, by more than 47% compared to Hadoop over IPoIB or GigE. WordCount, on the other hand, does not benefit much from Hadoop-A because of the small size of its intermediate data and low requirement on data movement, as shown by (c) and (d). We focus on TeraSort for the rest of the performance evaluation.

Figure 3.5(a) shows that MapTasks of TeraSort complete much faster with Hadoop-A, especially when the percentage of completion goes over 50%. This is because Hadoop-A only performs

light-weight operations such as fetching headers and setting up PQ, thereby leaving more resources such as disk bandwidth for MapTasks. Note that Hadoop reports the progress of ReduceTasks as soon as data is being merged. Hadoop-A implements the same. Because Hadoop-A waits until the completion of last MOF before merge, this results in seemingly slow progress of ReduceTasks in Hadoop-A. Hadoop-A still makes progress on ReduceTasks. Once it begins reporting, its progress in terms of percentage jumps up quickly, as shown by (b) and (d) for TeraSort and WordCount, respectively.

3.4.3 Dissection of ReduceTask Execution

Table 3.2: Time Breakdown of ReduceTask Execution (Seconds)

Slaves	Hadoop on GigE		Hadoop on IPoIB		Hadoop-A	
	Shuffle/Merge	Reduce	Shuffle/Merge	Reduce	PQ Setup	Pipeline
4	1238 (66.2%)	633 (33.8%)	1179 (65.7%)	615 (34.3%)	452 (42.5%)	613 (57.5%)
6	1066 (67.7%)	522 (32.3%)	1152 (65.7%)	602 (34.3%)	426 (45.5%)	511 (54.5%)
8	1016 (65.0%)	551 (35.0%)	1190 (65.0%)	641 (35.0%)	441 (44.2%)	556 (55.8%)
10	1137 (64.1%)	629 (36.0%)	1208 (65.1%)	649 (34.9%)	437 (44.6%)	543 (55.4%)
12	1143 (65.0%)	607 (35.0%)	1208 (65.4%)	639 (34.6%)	442 (45.1%)	538 (54.9%)
14	1194 (66.0%)	622 (34.1%)	1166 (65.6%)	612 (34.4%)	446 (45.3%)	539 (54.7%)
16	1182 (65.7%)	616 (34.2%)	1169 (64.9%)	631 (35.0%)	455 (47.6%)	501 (52.4%)
18	1192 (65.6%)	624 (34.4%)	1195 (65.6%)	628 (34.4%)	461 (45.7%)	547 (54.3%)
20	1158 (65.8%)	602 (34.2%)	1178 (65.7%)	614 (34.3%)	461 (46.3%)	535 (53.7%)
22	1164 (66.3%)	593 (33.7%)	1170 (65.8%)	608 (34.2%)	463 (45.2%)	563 (54.8%)
24	1150 (65.0%)	599 (35.0%)	1148 (65.9%)	597 (34.1%)	460 (47.4%)	509 (52.6%)

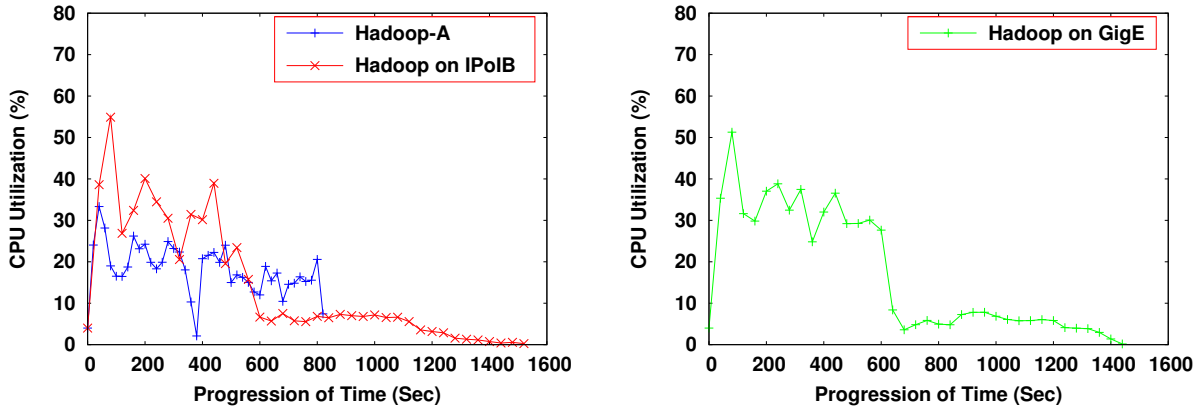
As shown in Figures 2.1 and 3.4, Hadoop-A avoids the serialization barrier between shuffle/merge and reduce phases of Hadoop ReduceTasks. Instead, it has a separate, light-weight phase to fetch headers and set up PQ. To shed light on how well the full pipeline of shuffle, merge and reduce in Hadoop-A benefits the performance of Hadoop, we run TeraSort with 4GB per ReduceTask and measure the time of different phases in Hadoop and Hadoop-A. The way that Hadoop and Hadoop-A maintain their execution statistics makes it possible and greatly simplifies this measurement. We collect timestamps at the begin and end of individual phases, and calculate the

elapsed time. The timestamps across different ReduceTasks are close to each other. We take the average across different ReduceTasks.

Table 3.2 shows our measurement results, including both the absolute time in seconds and the percentage of different phases during the execution of ReduceTasks. Note that the execution of ReduceTasks differs from that of the entire program by only a small duration, roughly the time taken to complete the first MOF. This can also be validated from Figures 2.1 and 3.4.

Hadoop-A significantly cuts down on the execution time of ReduceTasks. The shuffle/merge phase in Hadoop dominates the execution of ReduceTasks. Hadoop-A avoids shuffle/merge and performs only light-weight tasks such as header fetching and PQ setup. The PQ setup phase (including header fetching) is much faster compared to the shuffle/merge phase in Hadoop. Interestingly, even though Hadoop-A delays the start of merge until the completion of last MOF and overlaps the merge operation together with shuffle and reduce, the execution of the full pipeline for ReduceTasks can still be as much as 18% faster than the stand-alone reduce phase in Hadoop. This is mainly because our merge algorithm is levitated above local disks and avoids repetitive merges.

3.4.4 CPU Utilization



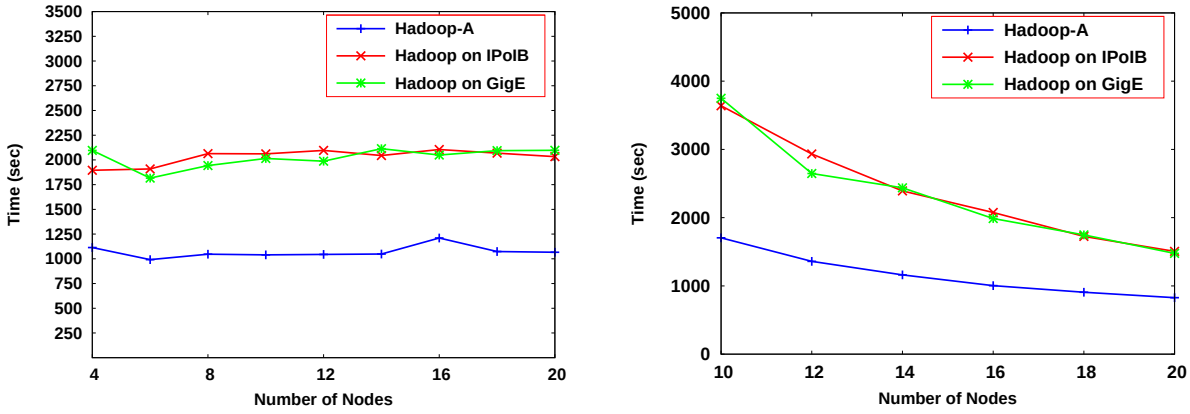
(a) Hadoop-A vs. Hadoop-IPoIB

(b) Hadoop-GigE

Figure 3.6: Comparison of CPU Utilization

We measure CPU utilization during the execution of TeraSort every 2 seconds. The percentage of CPU usage for 8 cores is recorded. We then take the average across all slaves at the same timestamp. Figure 3.6(a) shows the comparison of the average CPU utilization between Hadoop-A and Hadoop on IPoIB. Figure 3.6(b) shows that of Hadoop on GigE. These results are from a TeraSort program on 20 slave nodes. Similar comparisons are observed for TeraSort on different number of nodes. Clearly, Hadoop-A has less CPU utilization compared to Hadoop. Cumulatively, Hadoop-A has a CPU utilization of 18.7% at the time of its job completion, compared to 29.3% and 33.5% for Hadoop-IPoIB and Hadoop-GigE, respectively, at the same time point. Relatively, the reduction is 36.2% and 44.2%, respectively. Note that Hadoop-A has higher CPU usage towards the end of its completion, during which it is running a pipeline of shuffle/merge/reduce operations. The CPU utilization curve reveals that Hadoop-A is able to leverage system resource and sustain this pipeline, thereby shortening the execution time of TeraSort.

3.4.5 Scalability of Hadoop-A



(a) Weak Scaling.

(b) Strong Scaling.

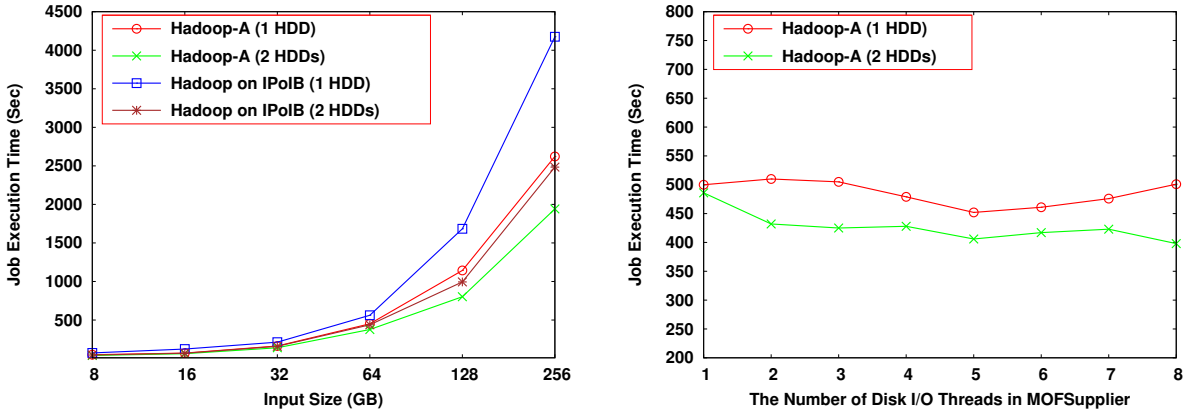
Figure 3.7: Hadoop-A Scalability.

Being able to leverage more nodes to process large amounts of data is an essential feature of Hadoop. We want to ensure Hadoop-A can deliver scalability in a similar manner. So we measure

the total execution time of TeraSort in two scaling patterns: one with fixed amount of total data (128GB) and increasing number of nodes, and the other with fixed data (4GB) per ReduceTask and increasing number of nodes. The aggregated throughput is calculated by dividing the total size with the program execution time.

Figure 3.7 (a) shows the scalability comparison between Hadoop-A and Hadoop with a fixed data size per node. Both Hadoop and Hadoop-A can achieve linear scalability. Hadoop-A can cut the execution time by approximately 50% and therefore double the throughput. Figure 3.7 (b) shows the scalability comparison between Hadoop-A and Hadoop with a fixed size of total data. Again both Hadoop and Hadoop-A can achieve good scalability. Hadoop-A can cut the execution time by up to 40% and 43%, compared to Hadoop on IPoIB and GigE, respectively. Conversely, this results in an throughput improvement of 66.7%, and 75.4%, respectively. These results adequately demonstrate better scalability of Hadoop-A for large-scale data processing compared to the original Hadoop.

3.4.6 Performance on Multiple Disks



(a) Increasing the Number of Disks

(b) Increasing the Number of I/O Threads

Figure 3.8: Tuning of I/O Performance

Slave nodes in a Hadoop cluster may be equipped with multiple disks, therefore Hadoop MapTasks and ReduceTasks are designed to utilize the bandwidth of all local disks. When several tasks

(either MapTask or ReduceTask) are running on the same node, their intermediate data are spread among all disks in a round robin manner. Thus the I/O traffic to any single disk can be greatly reduced. This can help shorten the wait time of I/O requests. For this reason, Hadoop-A is also implemented with multiple I/O threads to support data accesses to multiple disks. Accordingly, we have measured the performance of Hadoop and Hadoop-A when multiple disks are used to store the intermediate data.

Figure 3.8 shows the results of running TeraSort with different input sizes on 16 slave nodes. Increasing the number of disks can improve the performance of both Hadoop and Hadoop-A. When 2 disks are used for storing intermediate data, although the disk I/O bottleneck problem is significantly alleviated in Hadoop, Hadoop-A is still able to provide up to 21.9% better performance for 256GB data. In addition, we observe that the improvement of Hadoop-A increases with bigger data size. This is because bigger data size leads to more I/O requests, which causes a more severe I/O bottleneck at the disk. Therefore Hadoop-A can exploit benefits from the network-levitated merge algorithm.

In addition, we also notice that when multiple disks are used to store intermediate data, it is inefficient to use only one thread to read the data from all disks. This can cause underutilization of some disks while others are busy. Therefore we implement multiple I/O threads to serve the fetch requests in parallel. We measure the performance of Hadoop-A when several threads are used within the MOFSupplier. Figure 3.8(b) shows that multiple I/O threads can accelerate the processing of fetch requests and the improvement can be up to 18% when 8 threads are used. However, for tests with only one disk, increasing the number of threads has slightly degraded the performance. This is because the only disk is already overloaded. More threads actually introduce higher interferences among requests. Overall, our experiments demonstrate that Hadoop-A is capable of effectively utilizing multiple disks to improve the performance of Hadoop clusters.

It is worth noting that adding more disks improves the performance of Hadoop through reducing the disk I/O contention. This applies to additional investment in other resources as well

besides an investment on disks. However, our work is complementary to such hardware investments. Hadoop-A not only makes good use of the network, but also reduces the contention on disk bandwidth, thereby increasing the efficiency of I/O.

3.4.7 Improvement on Disk Accesses

Table 3.3: I/O Blocks

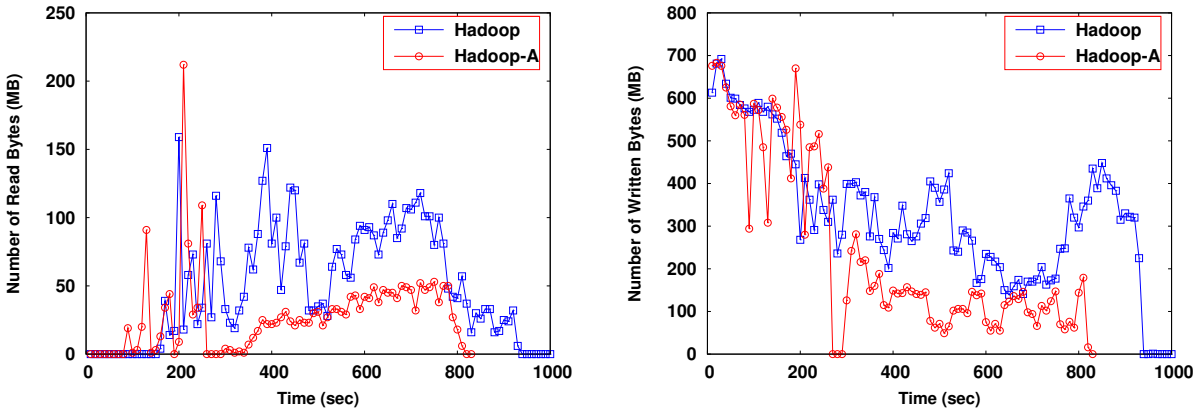
	READ (MB)	WRITE (MB)
Hadoop	5,426	36,427
Hadoop-A	2,441	22,713

Hadoop-A aims to lift the data shuffling and merging above disks for ReduceTasks through network-levitated merge algorithm. It avoids fetching the intermediate data to local disks. Instead, it leaves data on remote disks and only fetches small-size headers that can be stored in memory. In addition, the new merge algorithm sorts and merges all <key, val> pairs in memory. To assess the effectiveness of network-levitated merge, we have also measured the disk accesses during data shuffling under Hadoop-A, and compared the results with that of Hadoop. We run TeraSort on 20 slave nodes with 160GB as input size, each slave node has 4 MapTasks and 2 ReduceTasks. On each node we run *vmstat* and *iostat* to collect I/O statistics and trace the output every 2 seconds.

Table 3.3 shows the comparison of the number of bytes read and written by Hadoop and Hadoop-A into local disks per slave node. Overall, Hadoop-A significantly reduces the number of read blocks by 55.1% and write blocks by 37.6%. This effectively demonstrates that Hadoop-A reduces the number of I/O operations and relieves the load of underlying disks.

Figure 3.9 shows the progressive profile of read and write bytes during the job execution. During the first 200 seconds in which MapTasks are active, there is no substantial difference between Hadoop and Hadoop-A in terms of disk I/O traffic. After the first 200 seconds, ReduceTasks start fetching and merging the intermediate data actively. Because Hadoop-A uses the network-levitated merge algorithm which completely eliminates the disk access for the shuffling and merging of data

segments, we observe that Hadoop-A effectively reduces the number of bytes read from or written to the disks. Therefore, disk I/O traffic is significantly reduced during this period.



(a) Read Bytes

(b) Write Bytes

Figure 3.9: I/O Improvement

When disk bandwidth is a scarce resource, high disk I/O traffic can lead to long queuing time of I/O requests. This degrades the performance of the original Hadoop. In order to further analyze the benefit from the reduced disk accesses, we measure the service time and wait time of I/O requests for Hadoop and Hadoop-A. The service time is the time taken to complete one I/O request and the wait time includes an I/O request's queuing time and its service time. The result is shown in Figure 3.10. A couple of I/O behaviors can be observed from this figure. First, there is a big gap between Hadoop's service time and wait time, which indicates that most I/O requests have spent a huge amount of time waiting in the disk's queue. Second, the I/O service time is comparable between Hadoop and Hadoop-A. Figure 3.10 shows that Hadoop-A leads to similar or lower I/O wait time during the first 200 seconds, which corresponds to the mapping phase of the execution. As the execution progresses, the I/O wait time of Hadoop-A is significantly reduced when job enters into the shuffle/merge and reduce phases. This demonstrates that the reduction of disk accesses contributes to the reduction of I/O wait time. Taken together, these experiments

indicate that Hadoop-A with network-levitated merge can effectively improve the I/O performance in Hadoop, thereby effectively shortening job execution time.

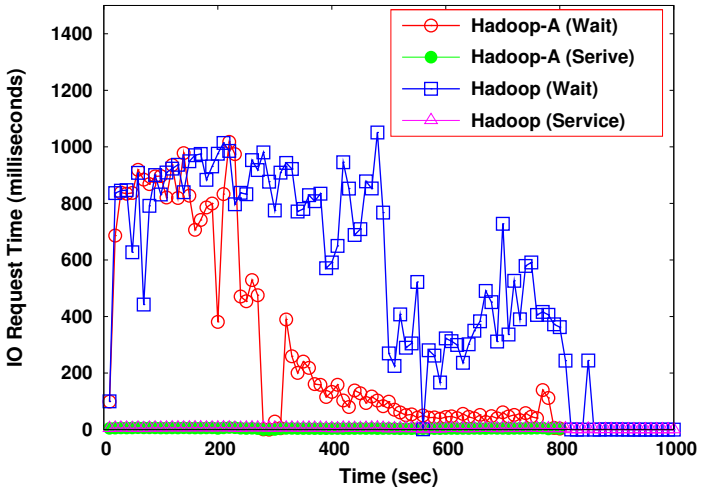


Figure 3.10: I/O Requests Service and Wait Time

3.4.8 Benefits of Merge Algorithm and RDMA

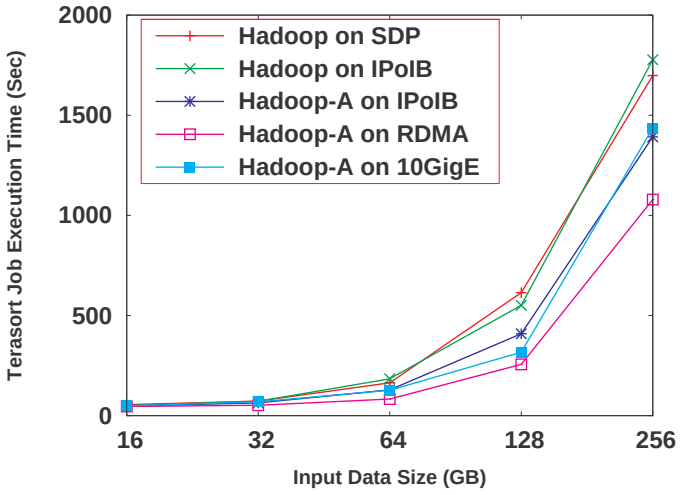


Figure 3.11: Performance benefit from Network Levitated Merge and RDMA, respectively

To investigate the respective improvement brought by Network Levitated Merge (NLM) and RDMA respectively, we compare the performance of Hadoop-A with different network protocols

on InfiniBand. When running on top of TCP/IP, performance improvement is mainly attributed to the NLM. We compare the performance of Hadoop-A when it is running on RDMA with that of running on IPoIB to quantify the improvement introduced by RDMA. The results are shown in Figure 3.11. As shown in the figure, on average, Hadoop-A on IPoIB efficiently reduces the job execution time by 18.9% when compared to Hadoop on IPoIB. This demonstrates that NLM is effective at improving the performance of Hadoop by reducing disk accesses on the ReduceTask side and forming a pipelined shuffle, merge and reduce phases. Figure 3.11 also shows that, by leveraging RDMA, Hadoop-A can further lower the job execution time. Compared to Hadoop-A on IPoIB, Hadoop-A on RDMA cuts down on the execution time by 19.9% on average. In addition, Figure 3.11 also shows that running Hadoop-A on 10 Gigabit Ethernet with TCP/IP achieves similar performance as Hadoop-A on IPoIB.

3.5 Related Work

MapReduce is a programming model for large-scale arbitrary data processing. The model popularized by Google provides very simple but powerful interfaces, while hiding complex details of parallelizing computation, fault-tolerance, distributing data and load balancing [33]. Its open-source implementation, Hadoop, provides a software framework for distributed processing of large datasets [3].

A rich set of research has been published on improving the performance of MapReduce recently. Originally, the Hadoop scheduler assumed that all nodes in a cluster were homogeneous and made progress with the same speed. Jiang et al. [48] conducted a comprehensive performance study of MapReduce (Hadoop), concluding that the total performance could be improved by a factor of 2.5 to 3.5 by carefully tuning the factors, including: I/O mode, indexing, data parsing, grouping schemes and block-level scheduling. Zaharia et al. [106] designed a new scheduling algorithm, Longest Approximate Time to End (LATE), for heterogeneous environments where ideal application environment might not be available. To fully take advantage of the multicore and multiprocessor systems, Ranger et al. [70] designed Phoenix, a programming API and runtime system

for shared-memory systems. In Phoenix, users only need to write simple parallel code without considering the complexity of thread creation, dynamic task scheduling, data partitioning, and fault tolerance across processor nodes. Considering the fact that original data structure used to group key/value pairs would be a primary performance bottleneck on the clusters of multicore architecture, Kaashoek et al. [62] designed a new MapReduce library with a compromised data structure, which outperforms its simpler peers, including Phoenix. Seo et al. [76] has leveraged prefetching and pre-shuffling techniques into MapReduce. They have shown that these techniques can improve the overall performance of Hadoop in shared environment. In [43], Sun engineers describe their work on running Hadoop over Lustre File System. Camdoop [32] is designed to decrease the network traffic caused by intermediate data shuffling through applying a hierarchical aggregation during the data forwarding. However, Camdoop is only effective in special network topology, such as 3D torus network, and its performance degrades sharply in common network topologies adopted by data centers.

The closest work to ours is MapReduce online as proposed by Condie et al. [30]. MapReduce online attempts to leverage push model to directly send the intermediate data from MapTasks to ReduceTasks to avoid touching disks on the MapTasks sides. In order to do so, it requires large number of sustained TCP/IP connections between MapTasks and ReduceTasks. However, it severely restricts the scalability of Hadoop MapReduce. In addition, when the data size is large and network cannot keep up with the MapTask processing speed, intermediate data still needs to be spilled to disks. Furthermore, it fails to identify the I/O bottleneck problem in HttpServlet and MOFCopier. So for the above reasons, MapReduce online has to fall back onto the original Hadoop execution mode. Our work addresses the similar performance issue of data movement, but differs from these studies by enabling network-levitated merge to avoid disk access and overlapping merge and reduce at ReduceTasks. Meanwhile, via continuing using request-driven, a.k.a pulling model, Hadoop Acceleration framework can maintain comparable scalability as original Hadoop MapReduce.

Leveraging RDMA from high speed networks for high-performance data movement has been very popular in various programming models and storage paradigms. Liu *et al.* [60] designed RDMA-based MPI over InfiniBand. Implementations of PVFS [66] on top of RDMA networks such as InfiniBand and Quadrics were described in [49] and [98], respectively. A recent evaluation [73] of Hadoop Distributed File system (HDFS) used the SDP [45] and IPoIB protocols of InfiniBand [**?**], and the authors showed that MapReduce is still unable to leverage the RDMA (Remote Direct Memory Access) communication mechanism available from high-performance RDMA interconnects such as InfiniBand and RoCE [41] (RDMA over Converged Ethernet). Our acceleration framework uses RDMA as its first communication protocol besides the TCP/IP protocol in the original Hadoop. Our work complements previous efforts to enable RDMA for Hadoop large-scale data processing programming model. Particularly, we show that RDMA is very beneficial in reducing Hadoop CPU utilization. Huang *et al.* [42] designed an RDMA-based HBase over InfiniBand. In addition, they also mentioned the disadvantages of using Java Socket Interfaces. Jose *et al.*[51, 50] implemented a scalable memcache through taking advantage of performance benefits provided by high-speed interconnects. Furthermore, Islam *et al.* [47] enhances the HDFS using RDMA over InfiniBand via JNI interfaces. Although Hadoop MapReduce is a fundamental basis of Hadoop ecosystem, there is lack of research on how to efficiently leverage high performance interconnects in Hadoop MapReduce.

3.6 Summary

In this chapter, we have examined the design and architecture of Hadoop's MapReduce framework in great detail. Particularly, our analysis has focused on data processing inside ReduceTasks. We reveal that there are several critical issues faced by the existing Hadoop implementation, including its merge algorithm, its pipeline of shuffle, merge, and reduce operations, as well as its lack of support for RDMA interconnects. Accordingly, we have designed and implemented Hadoop-A as an extensible acceleration framework that can allow plugin components to address all these issues. By introducing a new network-levitated algorithm that merges data without touching disks

and designing a full pipeline of shuffle, merge, and reduce phases for ReduceTasks, we have successfully accomplished an accelerated Hadoop framework, Hadoop-A. Our experimental results show that Hadoop-A doubles the data processing throughput of Hadoop, and also reduces CPU utilization by more than 36% by leveraging RDMA-based data movement.

Chapter 4

Preemptive ReduceTask based Fast Completion Scheduler

4.1 Introduction

To address the fairness issue revealed in Chapter 2.2 and ensure fast completion for jobs of various sizes, we design a combination of two techniques: the *Preemptive ReduceTask* mechanism and the *Fair Completion Scheduler*. Preemptive ReduceTask is a solution to correct the monopolizing behavior of long ReduceTasks. By enabling a lightweight working-conserving option to preempt ReduceTasks, Preemptive ReduceTask offers a mechanism to dynamically change the allocation of reduce slots. On top of this preemptive mechanism, the Fair Completion Scheduler is designed to allocate and balance the reduce slots among jobs of different sizes. In summary, we make the following contributions on the scheduling of jobs in data centers for fair and fast job completion in this chapter.

- We introduce the Preemptive ReduceTask mechanism for lightweight, work-conserving preemption, on top of which we design the Fair Completion Scheduler that improves both the fairness and execution efficiency of MapReduce jobs.
- We have conducted a systematic evaluation of Fair Completion Scheduler. Our results demonstrate that it can reduce the average execution time of workloads by up to 39.7% and improves the fairness by as much as 66.7%, when compared to Hadoop Fair Scheduler (HFS), meanwhile achieving significant performance improvement over Hadoop Capacity Scheduler (HCS).

4.2 Preemptive ReduceTask

A preemptive mechanism needs to be efficient and lightweight so that it can react fast enough to dynamic system workloads. But a ReduceTask often consumes the bulk of processing time

due to its main responsibilities of fetching and merging intermediate data from all MapTasks and performing user-defined reduce computation on the merged data. In this section, we introduce our Preemptive ReduceTask mechanism that can preempt a ReduceTask at any time during its execution, with low overhead and negligible delay to the job progress.

4.2.1 Work-Conserving Self Preemption

Preemption is usually an OS utility to threads and processes running on a system. Operating systems such as Linux are equipped with a sophisticated thread/process table along with virtual memory to record the progresses of threads/processes and support lightweight preemption. However, there is no such utility in Hadoop to keep the ReduceTask around as a process after its preemption. Although Hadoop currently provides a *killing* based preemption mechanism, our results show that killing is a poor preemption option that can significantly delay the progress of entire job. A naive checkpoint/start mechanism is also not suitable because it dumps all memory of a ReduceTask (it can be several GB) to persistent storage and incurs very high costs. Instead we introduce a work-conserving self preemption mechanism. When requested, a ReduceTask will conserve its work and then preempt itself, i.e., exit and release reduce slot. Note that our preemptive ReduceTask keeps current APIs of Hadoop and HDFS intact, so all existing Hadoop applications can still function without any modification.

During the shuffle phase, a ReduceTask fetches all the segments that belong to it from all intermediate map outputs. According to the sizes of the segments, ReduceTask stores them either to local disks or in memory. Meanwhile, multiple merging threads merge fetched segments into larger segments and store them to the persistent storage. During the reduce phase, a ReduceTask organizes all the segments in a *Minimum Priority Queue* (MPQ, which has a heap structure), in which the segment that has the minimum first <key,value> pair is positioned at the head of MPQ. As the reduce phase progresses, <key,value> pairs are continually popped out from the MPQ and supplied to the reduce function.

4.2.2 Preemption during Shuffle/Merge Phase

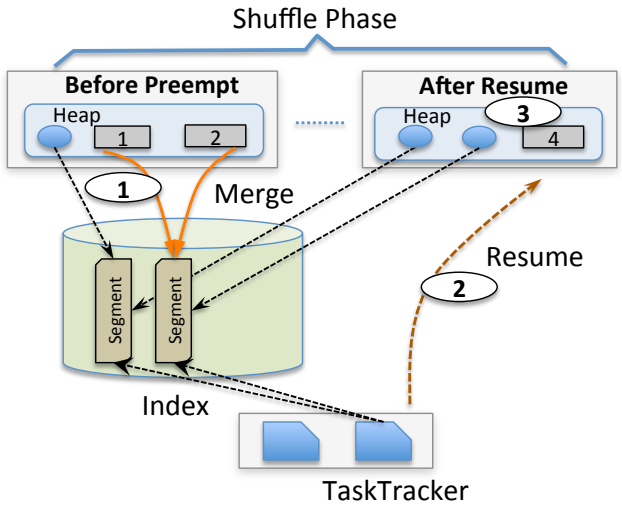


Figure 4.1: Preemption and Resumption during Shuffle/Merge Phase

Figure 4.1 shows our design of work-conserving preemption when a ReduceTask is in the shuffle phase. Before preemption, a ReduceTask has a mixture of one on-disk segment and two in-memory segments, organized in a heap. Preserving the state of shuffle phase is to keep track of the shuffling status of all segments. Upon receiving a preemption request, this ReduceTask merges the in-memory segments and flushes the results to the disks (Step 1) while leaves on-disk segments untouched. The parent TaskTracker maintains an index record on the locations of fetched segments, one per preempted ReduceTask. Then the ReduceTask preempts itself and releases the slot. When the ReduceTask is later resumed (Step 2), it retrieves the index record from the parent TaskTracker, then restores the heap structure before the preemption. After that, this ReduceTask continue to fetch the rest segments from remaining map outputs (Step 3).

4.2.3 Preemption during Reduce Phase

To conserve the work before preemption in the reduce phase, a ReduceTask needs to store the current results to HDFS besides recording the positions of input segments in the MPQ. In other words, ReduceTask needs to preserve the state of reduce computation at the end of each intermediate <key,val> pair, and remember the index of the last intermediate <key,val> pair at

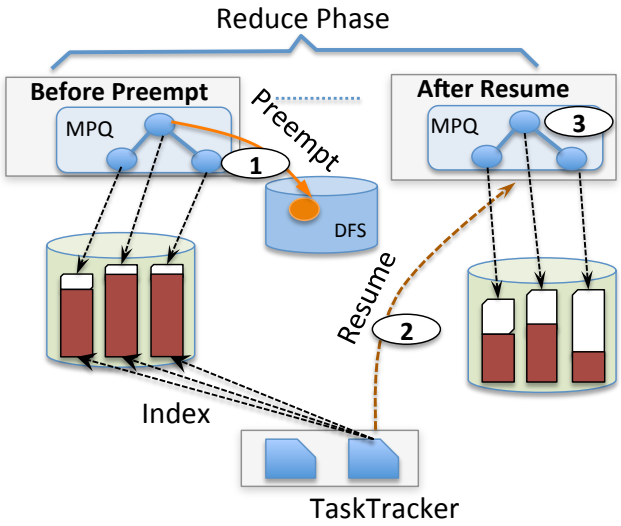


Figure 4.2: Preemption and Resumption during Reduce Phase

the time of preemption. Figure 4.2 shows our strategy for work-conserving preemption during the reduce phase. A ReduceTask is drawing <key,val> pairs from the MPQ that consists of three segments. When it receives a preemption request, it stops the reduce computation at the boundaries of <key,val> pairs (Step 1). Available results for previous <key,val> pairs are stored to HDFS. The parent TaskTracker again helps in this process by storing an index record for a preempted ReduceTask, and later provides it for preempted ReduceTask to resume its execution (Step 2). After resumption, the ReduceTask restores the MPQ again and proceeds further from the next <key,val> pair without any loss or repetition of reduce computation and intermediate data re-shuffling (Step 3). During this process, to allow multiple preempted/resumed ReduceTasks to write to the same HDFS files, we let TaskTracker maintain the output streams for the HDFS files, therefore they can be shared by many ReduceTasks with the same Id, and only the last ReduceTask closes the stream. In addition, Task migration is also possible for a preempted ReduceTask but it requires data to be re-fetched over the network.

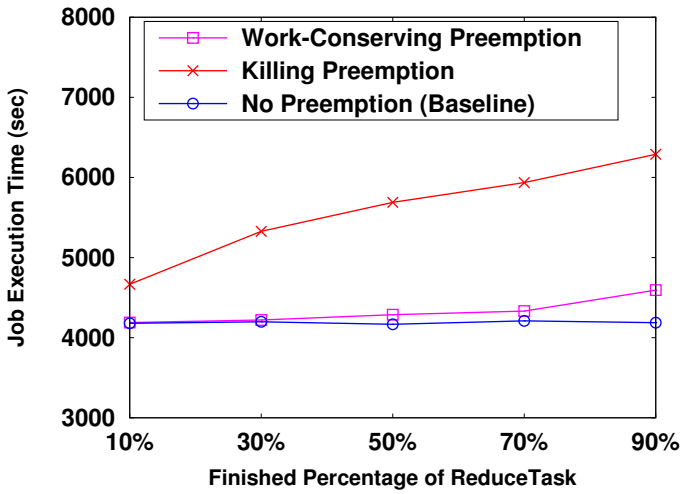


Figure 4.3: Measurements of Preemption Overhead

4.2.4 Work-Conserving Preemption Impact

To quantify the overhead of preemption at different points during the execution of ReduceTask, we employ a shuffle and reduce intensive *Terasort* benchmark [15], and run 512GB Terasort input data on 20 nodes (the average of 3 runs are reported).

In the experiment, we preempt every ReduceTask of a job at different stages and resume them after one heartbeat interval (3sec). The increases in the job execution times indicate the overhead imposed by these preemptions. Figure 4.3 shows that, compared to the execution without any preemption, our work-conserving mechanism can efficiently preempt and resume ReduceTasks with negligible overhead. The job execution time is kept nearly the same when ReduceTasks are preempted before their progress reaches 70% completion. Preempting ReduceTasks when they reach 90% progress causes noticeable overheads. This is because, when preemption happens at a later stage, storing larger partially completed results to HDFS can cause longer delay to the job progress. So in our current design, a ReduceTask is not preempted once its progress reaches 70% (c.f. the proposed scheduling algorithm in section 4.3). Also included in the comparison is the preemption based on existing *killing* mechanism. As shown in the figure, killing is a poor preemption option, it causes significant overhead even when ReduceTask is preempted at very early stage. We have also measured the impact of repetitive preemptions on job progress and

have observed similar results (not shown for conciseness here). These measurements adequately demonstrate that our work-conserving Preemptive ReduceTask is a viable lightweight preemption mechanism for task management.

4.3 Fair Completion Scheduler

Algorithm 1 FCS: Selecting ReduceTask to Preempt

```
1: L_running: {a list of running jobs of decreasing remaining work.}
2: J_i: {a job requesting new reduce slots.}
3: Demand(J_i) ← {J_i's demand for reduce slots.}
4: if Available_reduce_slots < Demand(J_i) then
5:    m ← Demand(J_i) - Available_reduce_slots
6:    for all j ∈ L_running ∧ IsPreemptable(J_j) ∧ (m > 0) do
7:       if (J_j.T_rs > J_i.T_rs) ∨ ((J_j.T_rs == J_i.T_rs) ∧ (J_j.R_left > J_i.R_left) then
8:          RL_n ← {J_j's list of running ReduceTasks}
9:          for all r ∈ RL_n ∧ (m > 0) do
10:            preempt r
11:            m ← m − 1
12:         end for
13:      end if
14:   end for
15: end if
```

To efficiently balance the reduce slots among a large number of jobs of different sizes, we introduce a novel preemptive ReduceTask scheduling policy via considering the remaining ReduceTasks workload of all the jobs. Meanwhile, MapTasks can be scheduled under independent scheduling policies, such as max-min fair, or FLEX [92]. Because of the benefits in achieving fairness for jobs of different sizes (c.f. Section 5), we refer to it as Fair Completion Scheduler (FCS).

As a preemptive scheduler, FCS must be equipped with two algorithms: one to automatically select a ReduceTask to preempt and the other to select a ReduceTask to launch. We first describe the selection policy for preemption. To select a suitable ReduceTask and achieve fair execution, we need to evaluate the run-time progress of jobs. However, the relative progress and the remaining

processing time of ReduceTasks are not available before they start. We choose the following approximations to estimate the progress.

Remaining shuffle time:

This is estimated as T_{rs} through the function: $T_{rs} = (\frac{M_{left}}{M_{rate}}) \times T_{mavg}$, where M_{left} stands for the number of remaining MapTasks, M_{rate} is the average rate in completing MapTasks, and T_{mavg} is the average execution time of MapTasks that have completed or in progress. As a job is making progress in its execution, we dynamically update M_{rate} accordingly.

Remaining reduce data:

This is estimated as R_{left} through the function: $R_{left} = R_{total} - R_{done}$, where R_{total} stands for the total intermediate data to reduce, and R_{done} the data that has been reduced. The latter is available during the progress of reduce phase, and the former is available when the reduce phase starts.

Execution Slackness:

This is estimated as E_{slack} through the function: $E_{slack} = \frac{T_{total}}{T_{est}}$, where T_{total} is a ReduceTask's total execution time since its beginning and T_{est} is its estimated execution time based on its progress without preemption. We calculate it as $T_{est} = \frac{T_{svc}}{C_{pctg}}$, where T_{svc} is the actual execution time excluding preemption and C_{pctg} is the percentage of completed work.

FCS is designed with policies to balance reduce slots between small jobs and large jobs. On the one hand, it compares a job that has the largest amount of remaining work to a job requesting reduce slots, as shown in Line 7 of Algorithm 1. Its ReduceTasks are preempted if it has more work than the requesting job (Line 10). Essentially, this allows small jobs to preempt large jobs, solving the monopolizing behavior of long-running jobs and reducing the delay of small jobs. On the other hand, we monitor the *execution slackness* of a ReduceTask since its beginning. If its execution slackness has reached a configurable upperbound (5 by default), a ReduceTask will not be preemptable, i.e. *IsPreemptable* returns false. This enables large jobs with an option to escape preemption–keeping their reduce slots–and avoid starvation. Note that the execution slackness is a calculated number at run-time, which offers a better choice than a static parameter, for example,

the number of times a ReduceTask can be preempted. Its sole purpose is to guarantee that a long job would not get seriously delayed because of frequent preemption by other jobs. Besides taking into account of execution slackness, we avoid preempting a newly launched ReduceTask or a ReduceTask whose progress has gone over 70% to avoid overhead.

Algorithm 2 FCS: Selecting ReduceTask to Launch

```
1:  {Receiving a heartbeat from node n with an empty slot.}
2:  L_rem: {a sorted list of jobs of increasing remaining work.}
3:  for all j ∈ L_rem do
4:     if (Task r is j's reduce task either preempted from n or never launched) then
5:        r.migration = 0
6:        launch r on n
7:        return
8:     end if
9:     T_prt ← {j's preempted ReduceTasks (oldest first)}
10:    for all r ∈ T_prt do
11:       if r.migration >D then
12:          migrate r to n
13:          r.migration = 0
14:          return
15:       end if
16:       r.migration += 1
17:    end for
18: end for
```

Then we describe briefly the policy for selecting a ReduceTask to launch, which is shown as Algorithm 2. In making this selection, FCS favors the jobs with the least amount of remaining work as shown in Line 2 of Algorithm 2. Jobs are firstly sorted according to their T_{rs} values, when two T_{rs} values are equal, they are sorted according to R_{left}. In addition, it takes the data locality into account, trying to launch a preempted ReduceTask on the same node that it has executed before (Line 4). A preempted ReduceTask that cannot achieve data locality will be delayed (Line 16). However, if a preempted ReduceTask has been delayed for too long because it is not able to resume on its previous node (Line 11), then FCS migrates it to another node that has available reduce slots (Line 12). In this algorithm, D is an approximation of $-M \times ln(\frac{1-L}{1+(1-L)})$, a similar parameter employed in the delay scheduling [102], where M is the number of nodes in the cluster

and L is the expected data locality. For example, on a cluster of 20 nodes, with the expected data locality $L = 0.95$, then $D \approx 61$. With this algorithm, we fit the same delay scheduling policy (and its parameter D) nicely into FCS, and delay the launching of a ReduceTask for a future possibility to resume it on the node it was preempted, i.e., better locality. This parameter allows us to consider the tradeoff between the need of resuming ReduceTask for data locality and the need of migrating ReduceTasks for free slot utilization. In Section 4.4.2, we show that careful tuning of D can indeed lead to a good tradeoff between these two factors.

4.4 Evaluation Results

This section presents a systematic performance evaluation of Fair Completion scheduler (FCS) using a diverse sets of workloads, including *Map-heavy* workload, *Reduce-heavy* workload, *Mixed* workload. Furthermore, we conduct stress tests through *Gridmix2* [8]. We compare the performance of FCS to the Hadoop Fair Scheduler (HFS) and Hadoop Capacity Scheduler (HCS). Several versions of Hadoop are available. Particularly, YARN as a successor of Hadoop provides a new framework for task management. However, through code examination and perform evaluation, we have found that YARN adopts the same task schedulers, thus facing the same fairness issues as Hadoop. In addition, YARN is still not yet ready for large-scale stable execution. Therefore, our evaluation is based on the stable version Hadoop 1.0.4.

4.4.1 Experimental Environment

Cluster Setup: Experiments are conducted in a cluster of 46 nodes. One node is dedicated as both the Namenode of HDFS and the JobTracker of the Hadoop. Each node is equipped with four 2.67GHz hex-core Intel Xeon X5650 CPUs, 24GB memory, and two 500GB Western Digital SATA hard drives.

Hadoop Setup: We configure 8 map slots and 4 reduce slots per node, based on the number of cores and memory available on each node. We assign 1024MB heap memory to each map and

reduce task, respectively. The HDFS block size is set to suggested 128MB [102] to balance the parallelism and performance for MapTasks. We use the best tuned configuration for our cluster.

Benchmarks: We employ the well-known *GridMix2* and *Tarazu* benchmarks [21] to demonstrate that FCS is suitable for various types of workloads.

Tarazu benchmarks represent typical jobs in production clusters. Meanwhile, Different benchmarks emphasize different workload characteristics. Map-heavy jobs generate a small amount of intermediate data, thus resulting in lighter ReduceTasks compared to the relatively heavier MapTasks. This group includes *Wordcount*, *TermVector*, *InvertedIndex* and *Kmeans*. On the other hand, Reduce-heavy jobs generate a large amount of intermediate data, thus causing heavy network shuffling and reduce computation at the ReduceTasks. This group includes *TeraSort*, *SelfJoin*, *SequenceCount*, and *RankedInvertedIndex*. We omit the description of GridMix2 here for saving space, but it worth mentioning that we configure the submission of GridMix2 jobs as a Poisson random process with a configurable arriving interval.

Evaluation Metrics: A number of performance metrics are used in our presentation. They are:

- ***Average execution time***: This is the plain average of execution time among a group of jobs, reflecting the efficiency of schedulers to a system.
- ***Maximum slowdown***: We refer to *slowdown* as the normalized execution time, which is defined earlier. *Maximum slowdown* is then the biggest slowdown among a group of jobs. This reflects the fairness to jobs of different characteristics.
- ***ReduceTask wait time***: It is defined as the time spent by a ReduceTask in waiting for reduce slots after the same job's MapTasks (i.e. the entire map phase) have all completed. If the ReduceTask gets a slot before that, then the wait time is 0. This aims to reflect the delay experienced by ReduceTasks.
- ***Average preemption times***: This is the average number of preemptions experienced by a group of jobs with similar job sizes. This quantifies the distribution and frequency of preemptions to jobs of different groups that differ in job sizes.

4.4.2 Evaluating Design Choices of FCS

The design of FCS includes a couple of important design choices such as the threshold parameter that allows task migration to resume a preempted ReduceTask, and the choice of Preemptive ReduceTask instead of killing as the preemption mechanism. In this section, we conduct experiments to evaluate these design choices and elaborate their importance.

Opportunistic ReduceTask Migration

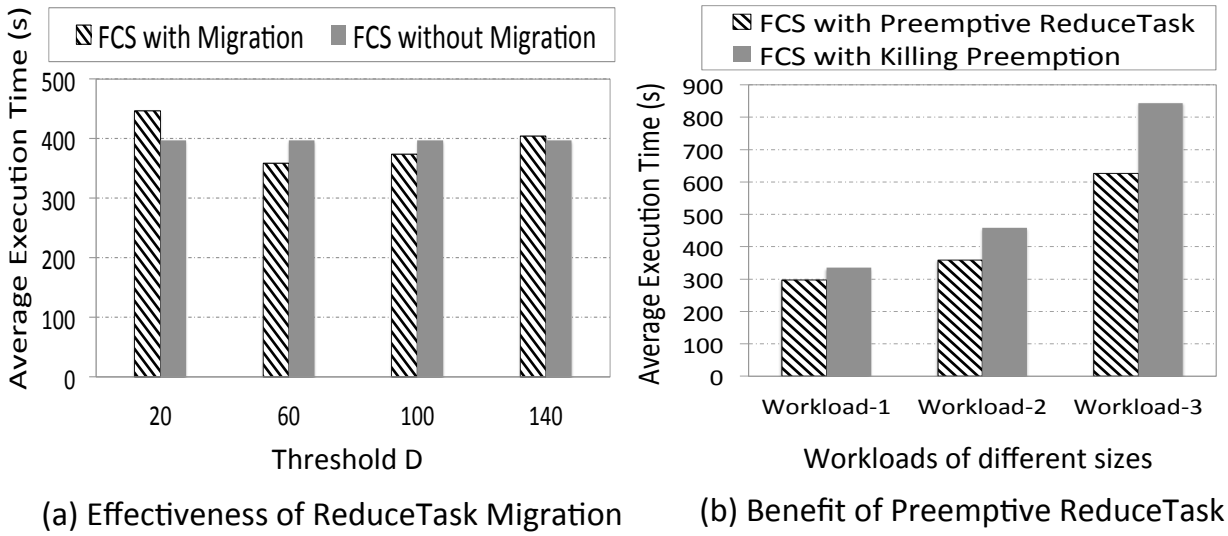


Figure 4.4: Evaluation of Design Choices

As mentioned in section 4.3, FCS is designed with an opportunistic parameter *D* that controls the tradeoff between keeping ReduceTasks on their original node for data locality and migration ReduceTasks to other available slots for resource utilization. A very large *D* allows a ReduceTask to be delayed many times and become sticky to their original nodes, achieving better data locality for the resumed ReduceTask but at the cost of underutilization of other reduce slots. In contrast, a very small *D* leads to better resource utilization but also incurs more data movement. In this section, we assess the impact of *D* by executing a pool of Gridmix2 jobs. Also, job submission is configured to follow a Poisson random process with an average inter-arrival time of 30 seconds.

In the experiment, we increase the *D* from 20 to 140, and compare the performance results of FCS with migration to that of FCS without task migration. As shown in the Figure 4.4(a), FCS with migration can lead to the best average execution time when *D* equals 60, with an improvement of 9.6%. Neither a small D of 20 or a large D of 140 can achieve a good balance between data locality and resource utilization. This experiment confirms that opportunistically allowing task migration as controlled by *D* can lead to good system performance. In the following sections, we use 60 as the value for *D*.

Benefits of Preemptive ReduceTask

We investigate the efficiency of FCS when preemption is enabled with either the Preemptive ReduceTask or the killing-based approach. We use three GridMix2 workloads of different numbers of jobs (80 for Workload-1, 130 for Workload-2 and 180 for Workload-3). Figure 4.4(b) shows the results. Compared to FCS with killing-based preemption, FCS with Preemptive ReduceTask effectively reduces the average execution time by 11.3%, 21.8% and 25.7% for three Workloads respectively. This demonstrates that FCS performs more efficiently with Preemptive ReduceTask than with the killing approach. So in the rest of this paper, we focus on further evaluation of FCS with the Preemptive ReduceTask.

4.4.3 Results of Map-heavy Workload

We now present the evaluation results on Map-heavy workload. The workload composition is shown in Table 4.1, featuring two basic characteristics. First, as shown in empirical trace studies [29, 53] that realistic workloads exhibit a heavy-tailed distribution for job sizes, translated into the number of MapTasks in the job. Second, to capture the effect that jobs arrive to the MapReduce cluster according to a random process, their arrival interval follows a Poisson random process with an average inter-arrival time of 30 seconds. For ease of presentation, we sort the jobs according to their input sizes and the requested number of tasks, then divide them into 10 different groups

Table 4.1: Job Composition of Map-heavy Workload

Group	Benchmark	Maps	Reduces	Jobs
1	WordCount	10	1	50
2	TermVector	20	2	40
3	InvertedIndex	50	4	30
4	TermVector	100	8	20
5	Kmeans	500	10	10
6	TermVector	1000	20	8
7	Kmeans	5000	20	6
8	InvertedIndex	10000	60	4
9	TermVector	15000	120	2
10	InvertedIndex	20000	180	1
			Total Jobs	171

Table 4.2: Performance of Map-heavy Workload

In Seconds	FCS	HFS	HCS
Average Execution Time	247	359	1061

of increasing sizes. This categorization means to help understand the scheduling effects on jobs of different sizes.

Table 4.2 shows the average execution time for all jobs in Map-heavy workload with different schedulers. Both FCS and HFS significantly outperform HCS, which groups jobs into a small number of job queues, within each queue, HCS adopts FIFO scheduling policy that is known to bias against small jobs and cause long average execution times. Thus we focus on the comparisons between FCS with HFS in the rest performance tests on Map-heavy workload. Overall, FCS speeds up the average execution time by 31% compared to HFS.

To shed light on how FCS treats jobs of different sizes, we crystallize the average execution times for the 10 different job groups inside workload. Figure 4.5 shows that FCS effectively reduces the average execution time for 9 different groups compared to HFS, achieving up to 2.4$\times$ speedup for jobs in group 2. Only jobs in Group 9 are negatively affected by FCS, at an average ratio of 0.79.

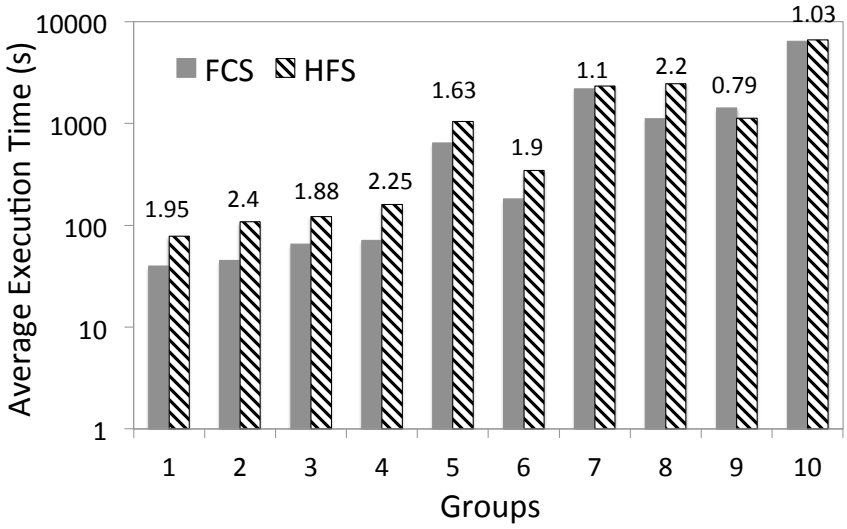


Figure 4.5: Average Execution Times of Jobs in Different Groups of Map-heavy Workload

We also perform an analysis on how these jobs are completed by the system over time. Figure 4.6 shows the CDF (Cumulative Distribution Function) of job completion for the workload grouped into three subsets: small (Groups 1-4), medium (Groups 5-7) and large (Groups 8-10). As shown in the figure, FCS significantly accelerates the completion of small jobs, moderately improves the jobs of medium sizes, and also helps large jobs reach higher completion rate at the beginning. In contrast, HFS severely delays above 15% small jobs in Groups 1-4 with a long distribution tail stretching beyond 600 seconds. These results again confirm that large jobs can monopolize reduce slots under HFS, depriving small jobs of a good share. FCS corrects such behavior in general, and helps small jobs in particular without much degradation for large jobs in the Map-heavy workload.

FCS improves system performance by mitigating the starvation of small jobs. It prioritizes jobs whose shuffle phases are about to complete, thus reducing the ReduceTask wait times. Figure 4.7 demonstrates the efficiency of FCS via showing the average ReduceTask wait time for all jobs in 10 groups. As we can see, the average wait time is dramatically cut down for the first eight groups by as much as 32.2$\times$ for group 5. Only for the last two groups, the wait times are stretched slightly.

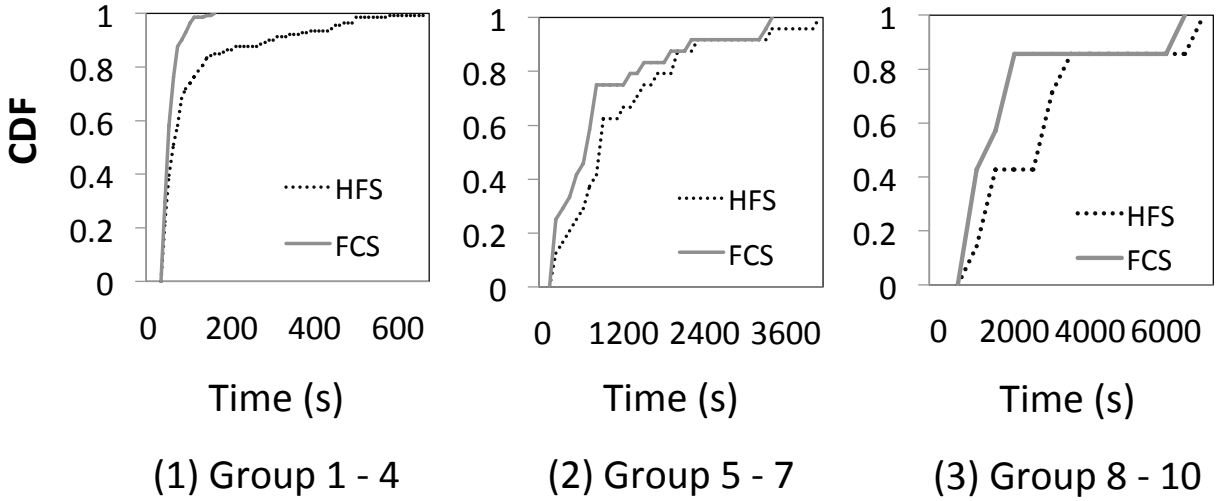


Figure 4.6: CDF of Job Completion Times of Jobs in Different Groups of Map-heavy Workload

To further obtain insights on how FCS has triggered preemptions to different jobs, we record the preemptions experienced by all ReduceTasks. Figure 4.8(a) shows the distribution of preemptions to different groups of jobs in the workload. As shown in the figure, preemptions have not happened to Groups 1-4. Groups 5-10 have experienced a small number of preemptions. This corroborates that FCS can be effective in delivering fair and fast completion without imposing excessive preemptions.

We have measured the maximum slowdown of all jobs to evaluate the fairness of schedulers to different jobs. As shown in Figure 4.8(b), FCS efficiently improves the fairness by 66.7%, compared to HFS, and achieves nearly uniform maximum slowdown across 10 groups. In contrast, HFS causes serious unfairness to small jobs. In the worse case, a job in Group 3 is slowed down by 16 times.

4.4.4 Results of Reduce-heavy Workload

Map-heavy workload represents jobs that generate small amount of intermediate data. In this section, we continue our evaluation with Reduce-heavy workload, in which jobs generate a large amount of intermediate data, resulting in long running ReduceTasks. The ratio of intermediate data

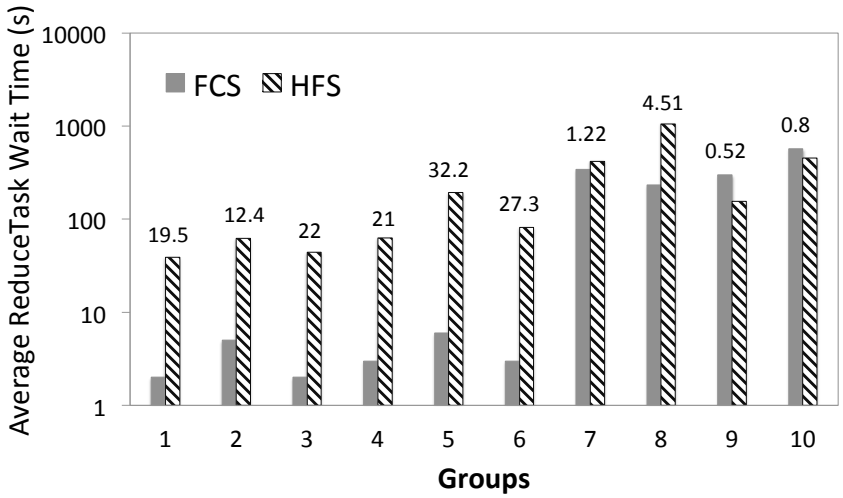


Figure 4.7: Average ReduceTask Wait Times of Jobs in Different Groups of Map-heavy Workload.

size to input size of those jobs is from 1 : 1 to 3 : 1. The job composition in the workload is listed in Table 4.3. We adopt the same distributions for job sizes and their arrival times as described in section 4.4.3.

We conduct the same set of experiments for Reduce-heavy workload as done for the Map-heavy workload to demonstrate that FCS can schedule different workloads effectively. For succinctness, we avoid redundant description, omit some figures, and only highlight the differences. Table 4.4 shows the overall performance under three schedulers. FCS speeds up the average execution time of the workload by 28% when compared to the HFS, and HCS still performs worse than the other two.

Figure 4.9 shows that FCS speeds up the average execution times of all 10 different groups in the workload. This differs from the Map-heavy workload. In addition, CDFs in Figure 4.10 show that FCS improves the completion rate for all three sets of job groups. Different from the experiment for the same set jobs in Map-heavy workload, FCS even delivers better improvements to large jobs in Groups $8-10$ of Reduce-heavy workload. This is because in Reduce-heavy workload, map phases of large jobs run much longer to generate intermediate data. When small jobs arrive, they preempt those long running ReduceTasks whose jobs are still mainly in the map phases. As a

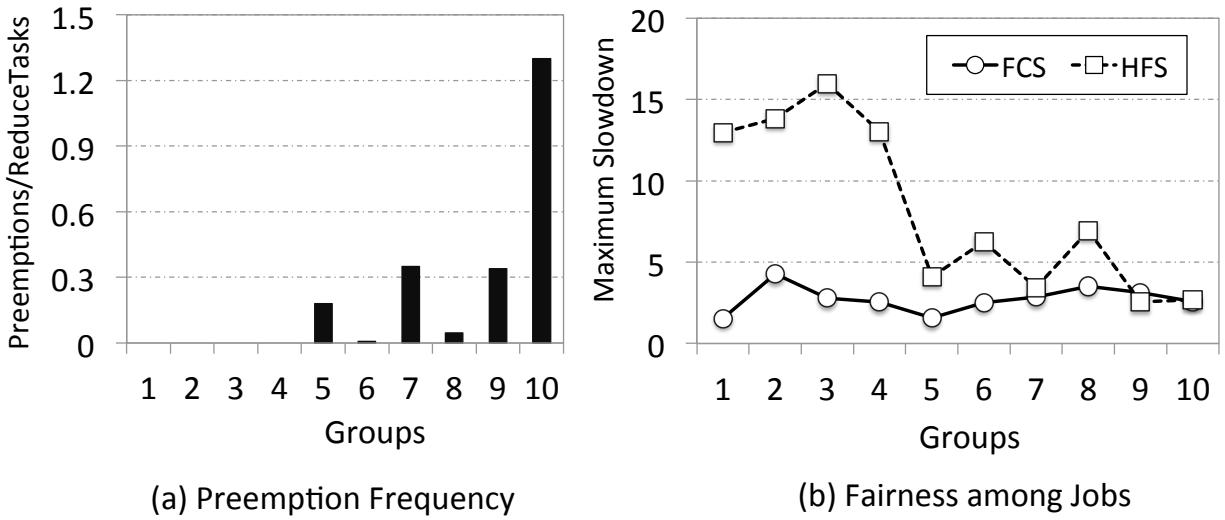


Figure 4.8: Measurement of Preemption Frequency and Fairness among Jobs in Map-heavy Workload

result, such preemptions cause little performance impact on large jobs, but efficiently accelerate the small jobs. As small jobs quickly leave the cluster, large jobs obtain more resources to achieve faster job completion. Thus we observe faster large jobs.

Significantly shortened ReduceTask wait time contributes to the fast job completion. Figure 4.11(a) compares the average ReduceTask wait times between FCS and HFS. For Groups 9 and 10, FCS leads to a slightly longer delay, up to 15%. For Groups 3 and 8, FCS and HFS are comparable. Group 1 has zero wait time in both cases. FCS drastically reduces the wait time down to 0 for Groups 2,4, and 5. For Groups 6-7, FCS efficiently cuts down the average ReduceTask wait time, thus speeding up the job completion. Figure 4.11(b) shows that FCS not only reduces the average execution time but also efficiently improves the fairness by 35.2% on average when compared to the HFS for the Reduce-heavy workload.

4.4.5 Results of Mixed Workload

Lastly, we generate two mixed workloads using all of the benchmarks from Tarazu. We omit the job composition table here and directly shows the results for saving space. Different from previous workloads, we derive our mixed workloads from the Facebook and Microsoft Bing traces

Table 4.3: Job Composition of Reduce-heavy Workload

Group	Benchmark	Maps	Reduces	Jobs
1	TeraSort	10	2	50
2	SelfJoin	20	4	40
3	SequenceCount	50	8	30
4	TeraSort	100	16	20
5	SelfJoin	500	32	10
6	RankInvertedIdx	1000	64	8
7	TeraSort	5000	128	6
8	SequenceCount	10000	256	4
9	TeraSort	15000	512	2
10	SequenceCount	20000	1024	1
			Total Jobs	171

Table 4.4: Performance of Reduce-heavy Workload

In Seconds	FCS	HFS	HCS
Average Execution Time	978	1364	8829

described in [24] to mimic the jobs on real production clusters. These two sets of workloads have different distributions: the Facebook workload consists of mostly small jobs at a percentage of 85%, while Bing workload contains more medium and large size jobs (57%).

Here, we only include some results for a succinct representation. Table 4.6 shows that compared to HFS, FCS reduces the average execution time by 20.2% for Facebook workload, and 34.8% for Bing workload. CDFs in Figure 4.12 further reveal that FCS speeds up small jobs in the Bing workload more efficiently. For the Facebook workload, FCS provides moderate improvements, since small jobs have a dominant presence in the workload with very few large jobs submitted very late, small jobs' requirement for reduce slots can be quickly satisfied, leaving less room for our scheduler to improve. Figure 4.13 again demonstrates that FCS can enhance the fairness by 41.1% for Facebook workload, and 46.4% for Bing workload. In both cases, the variation of maximum slowdown is much higher under HFS than under FCS due to the unpredictability of incoming jobs in terms of both job size and arrival rate. We also observe fluctuated slowdown among different groups of jobs under FCS, we deem it as a natural dynamic behavior of scheduler when dealing with a pool of randomly-arrived jobs.

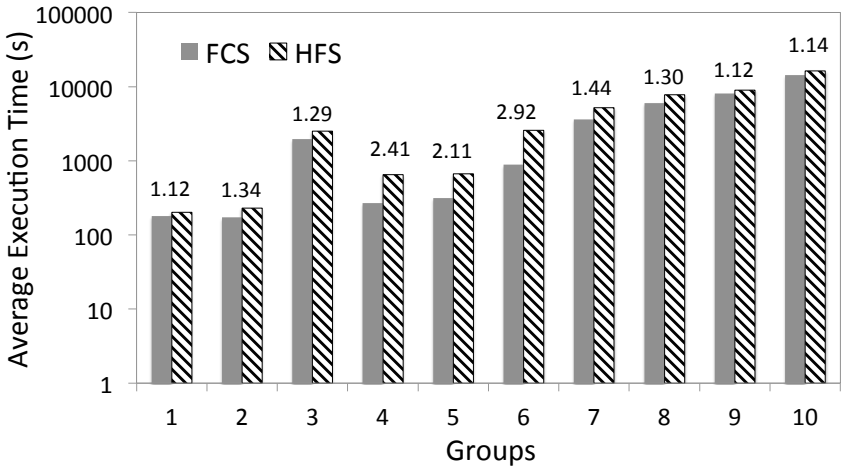


Figure 4.9: Average Execution Times of Jobs in Different Groups of Reduce-heavy Workload

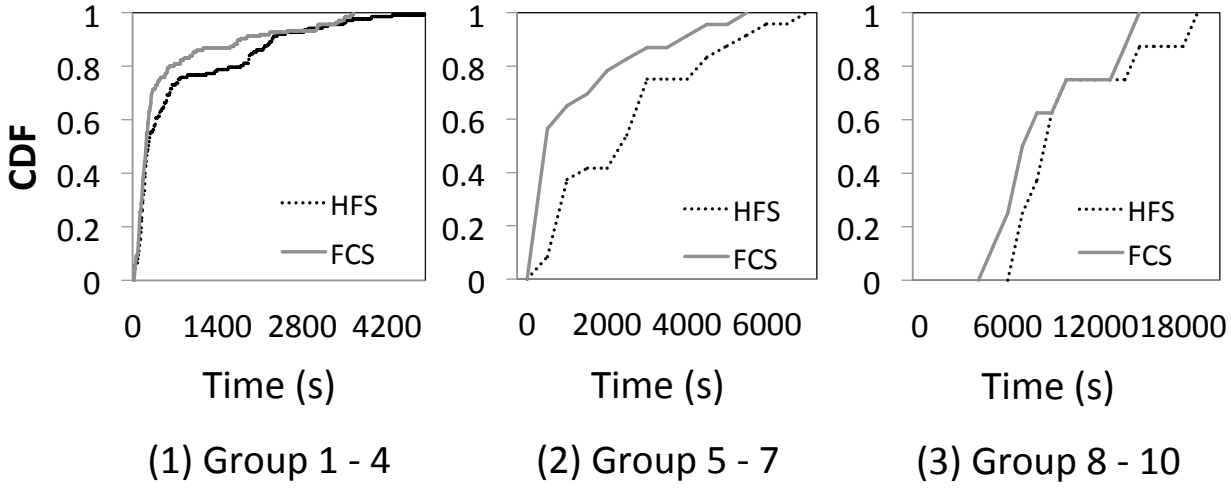


Figure 4.10: CDF of Job Completion Times of Jobs in Different Groups of Reduce-heavy Workload

4.4.6 Evaluation of Scalability

The workloads submitted to a production cluster varies over different periods of time. The capability of efficiently scheduling a large number of random arriving jobs is critical for a Hadoop scheduler, especially when the system is heavily loaded. We evaluate the scalability of FCS by varying the number of GridMix jobs from 60 to 300 and maintain the same distribution of job sizes. The experimental results are shown in Figure 4.14. Compared to HFS, FCS consistently reduces the average execution times of different experiments. On average, FCS reduces the average execution time by 39.7%. More importantly, FCS shows stable performance improvement when

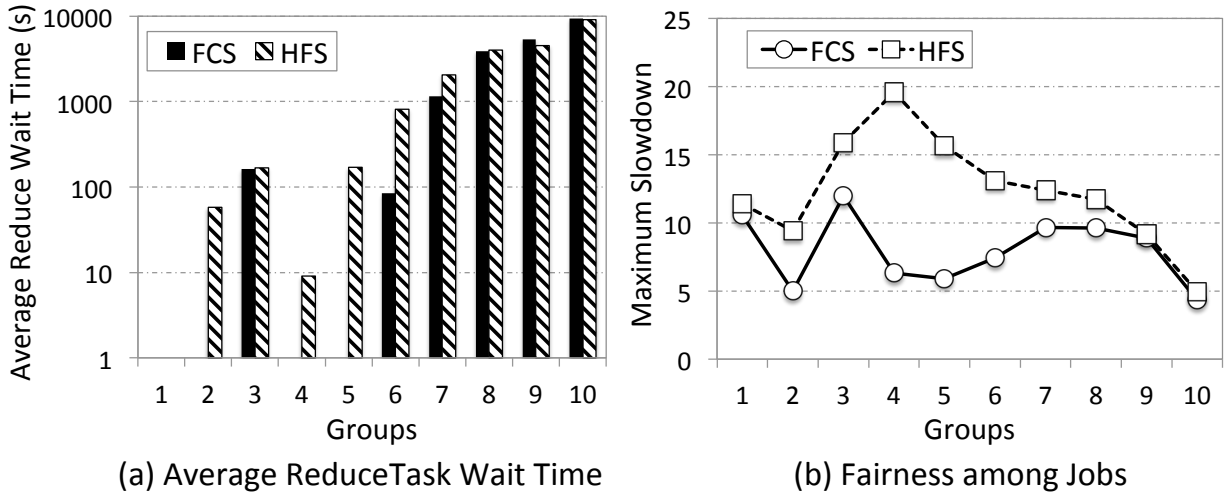


Figure 4.11: Measurement of Average ReduceTask Wait Time and Fairness among Jobs in Reduce-heavy Workload

the number of jobs increases. Furthermore, when the number of jobs increases, no noticeable scheduling overhead is observed in the JobTracker.

4.5 Related Work

Many MapReduce schedulers have been proposed over the past few years trying to maximize the resource utilization in the shared MapReduce clusters. Zaharia *et al.* introduced delay scheduling [102] that speculatively postpones the scheduling of the head-of-line tasks and ameliorate the locality degradation in the default Hadoop Fair scheduler [10]. In addition, Zaharia also proposed Longest Approximate Time to End (LATE) [106] scheduling policy to mitigate the deficiency of Hadoop scheduler in coping with the heterogeneity across virtual machines in a cloud environment. But either of these two scheduling policies supports task preemption for jobs in the same pool, thus unable to correct the monopolizing behavior of long-running ReduceTasks. Mantri [**?**] was designed to mitigate the impact of outliers in MapReduce cluster, it monitors task execution with real-time remaining work estimation, and accordingly take measures such as restarting outliers, placing tasks with network awareness and conserving valuable work from the tasks. But Mantri does not identify the resource monopolizing issue among large number of concurrent jobs

Table 4.5: Job Composition of Mixed Workloads

Group	Benchmark	Maps	Reduces	Facebook	Bing
1	WordCount	8	1	28	20
2	TermVector	8	2	28	12
3	InvertedIndex	8	2	29	11
4	SelfJoin	30	16	2	4
5	Kmeans	30	20	2	4
6	TeraSort	110	40	4	12
7	RII	100	50	4	12
8	SC	248	128	1	12
9	TermVector	300	180	1	11
10	SC	3600	256	1	2
			Total Jobs	100	100

Table 4.6: Performance of Mixed Workloads

	Facebook		Bing	
	FCS	**HFS**	**FCS**	**HFS**
Average Execution	95	119	531	814

caused by long-running ReduceTasks and does not provide lightweight preemption solution. Ahmad [21] proposed communication-aware placement and scheduling of MapTasks and predictive load-balancing for ReduceTasks as part of Tarazu to reduce the network traffic of Hadoop on heterogeneous clusters. But it also does not address the fairness and monopolization issues. Isard *et al.* [46] introduced the Quincy scheduler, which adopts min-cost flow algorithm to achieve a balance between fairness and data locality for the Dryad. But their use of killing as preemption mechanism can cause significant resource waste. Verma [87] introduced ARIA to allocate appropriate amount of resources to MapReduce job so that it can meet SLO. Based on ARIA, Zhang *et al.* [107] further studied the estimation of required resources for completing a Pig program to meet SLO. Lama [56] proposed AROMA to automatically determine the system configuration for Hadoop jobs to achieve quality of service goal. FLEX [92] aims to optimize different given scheduling metrics based on a performance model between slots and job execution time. However, none of

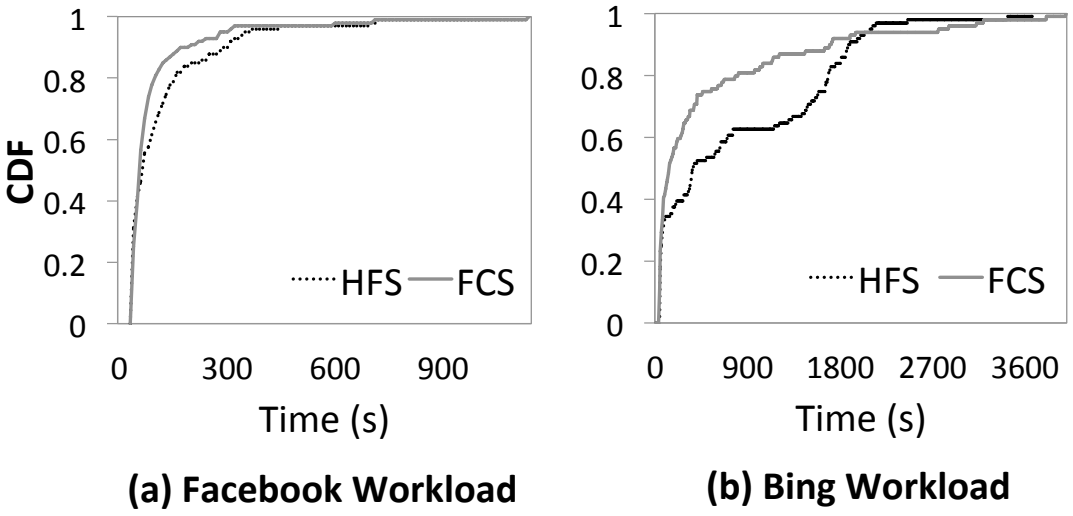


Figure 4.12: CDF of Mixed Workloads

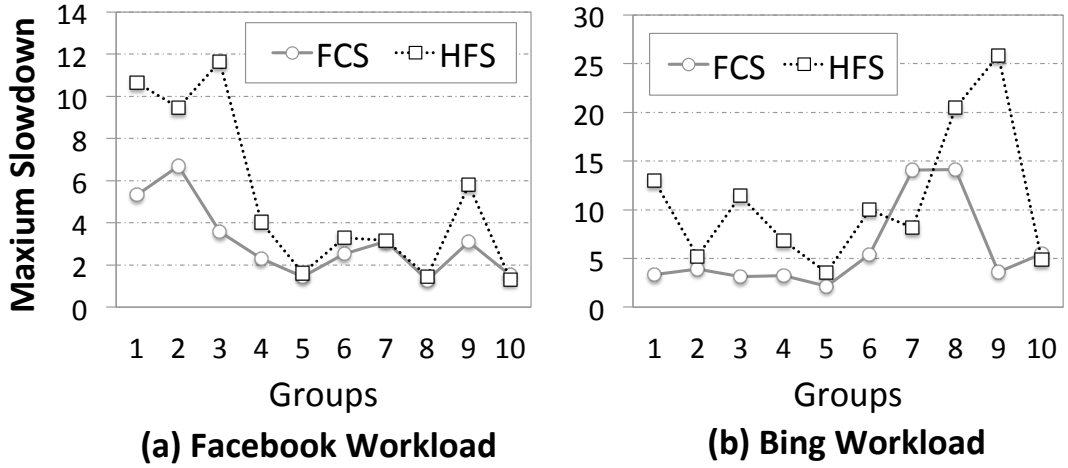


Figure 4.13: Fairness of Mixed Workloads

above four work considers the resource contention issue (reduce slot contention) among continuously incoming jobs in shared MapReduce clusters. In [22], Ananthanarayanan proposed Amoeba which supports lightweight elastic tasks that can release the slots without losing previous I/O and computation. This bears strong similarity to our preemptive ReduceTask. However, it imposes many constraints such as safe points on task processing so that tasks can be interfered without losing previous work. However no overhead measurement is reported in the article. In addition, no corresponding scheduling policy is designed to leverage the benefits provided by elastic task.

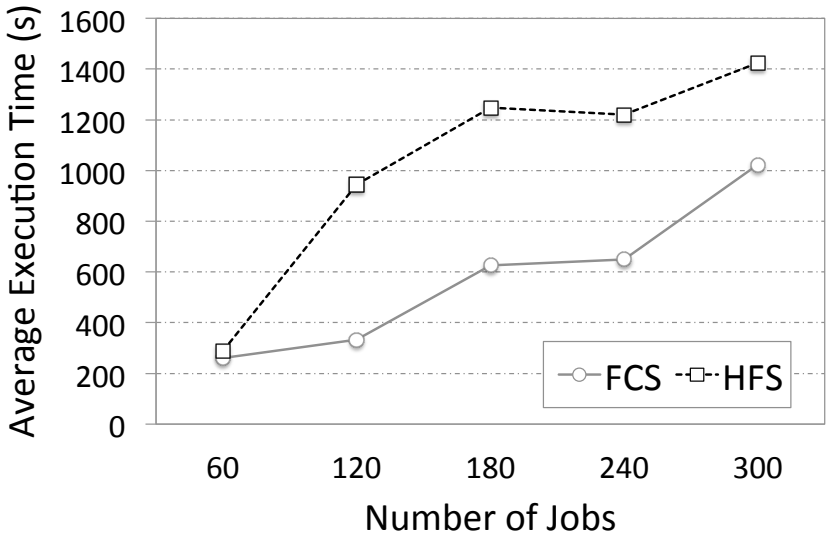


Figure 4.14: Scalability Evaluation with GridMix2

Recently, YARN [13] has been proposed by Yahoo! as the next generation MapReduce. It separates the JobTracker into ResourceManager and ApplicationManager, and removes task slot concept. Instead, it adopts resource container concept that encapsulates the general resources, such as memory, CPU and disk I/O into the schedulable unit (current YARN only supports memory). But our initial evaluation discovers that monopolization behavior of long-running ReduceTasks still exist in such framework as long as schedulers greedily allocate as many resources as permitted to one job. Therefore, our Preemptive ReduceTasks and Fair Completion Scheduler can be very beneficial in the new framework. In future, we plan to incorporate our techniques into the YARN.

4.6 Summary

In this chapter, we have revealed that there exists a serious fairness issue for the current MapReduce schedulers due to the lack of a lightweight preemption mechanism for ReduceTasks. Accordingly, we have designed and implemented the Preemptive ReduceTask as a work-conserving preemption mechanism, on top of which we have designed the Fair Completion Scheduler. The introduction of the new preemption mechanism and the novel ReduceTask scheduling policy have solved the fairness issue to small jobs, resulting in improved resource utilization and fast average

job completion for all jobs. Our design of Fair Completion Scheduler, compared to the Hadoop Fair Scheduler and Capacity Scheduler, can reduce the average job execution time by up to 39.7% and 88.9%, respectively. Furthermore, the Fair Completion Scheduler improves the fairness among different jobs by up to 66.7%, compared to the Hadoop Fair Scheduler.

Chapter 5

Enhancing the Adaptability of MapReduce for HPC Platforms

5.1 Introduction

Recall from section 2.3 that MapReduce and HPC systems exhibit distinct design paradigms. These distinctions between compute- and data-centric paradigms have significant performance implications to different types of MapReduce workloads. Therefore, it is imperative to characterize the performance of key architectural components in these two different paradigms. In this chapter, we aim to answer how does the configuration of storage resources such as parallel file systems affect job scalability and throughput? What is the impact of data placement and task scheduling? And how to reconcile and converge the architectural differences between the two paradigms so that one system can be configured and tuned for productive sharing by both conventional HPC applications and the emergent MapReduce-based analytics applications. Meanwhile, we can enhance the adaptability of MapReduce frameworks on HPC platforms.

In this chapter, we undertake an effort with intensive experiments to characterize the performance, identify the inefficiency of a MapReduce-based framework on the compute-centric paradigm, and compare its performance with that on the data-centric paradigm. Accordingly, we also introduce several optimizations targeting at compute-centric HPC systems.

Among many MapReduce frameworks, we have chosen Spark [103], which is a memory-resident implementation shown to outperform Hadoop for many applications by orders of magnitude [103, 93]. We leverage the Hyperion [11] system at Lawrence Livermore National Laboratory with two distinct configurations: one under the compute-centric paradigm and the other under the data-centric paradigm.

In summary, we conduct a comprehensive investigation to characterize the performance critical aspects of compute- and data-centric paradigms and shed light on how to build a dual-purpose

HPC system to enable fast data analytics. We have made the following contributions in this research.

- We have studied the impact of storage architecture to the performance of different types of MapReduce jobs, and revealed that their performance on HPC systems is highly dependent on their computation intensity.
- We have characterized the importance of intermediate data placement and the benefits of hierarchical storage media to Spark applications. Particularly, we show that MapReduce applications need to be aware of the performance implications of storage consistency mechanisms on HPC systems and avoid the cascading effects of lock contention from HPC file systems such as Lustre.
- We have evaluated the impact of locality-oriented scheduling techniques for MapReduce jobs on compute-centric HPC systems. We show that maximizing data locality is not so critical, and delay scheduling [102], a popular strategy to delay tasks for data locality can even cause performance degradation.
- We have introduced two optimization techniques: Enhanced Load Balancer and Congestion-Aware Task Dispatching. The former takes into account performance variation and imbalanced data distribution when scheduling tasks, resulting an improvement of 26% on job execution time. The latter recognizes the existing oblivity of Spark to new storage devices such as SSD, throttles the launch of Spark tasks and mitigates the congestion, thereby achieving a performance gain up to 41.2%.

5.2 Comparison Between Compute- and Data-Centric Paradigms

In this section, we aim to provide a direct comparison between the compute-centric and data-centric processing paradigms. We have used HPC systems to represent the compute-centric

paradigm, while using Spark, introduced in Section 1.1.3, as a representative framework of Data-Centric Paradigm. However, to avoid repeating the description in Section 1.1.3, we omit its description here.

5.2.1 HPC Systems Representing the Compute-Centric Paradigm

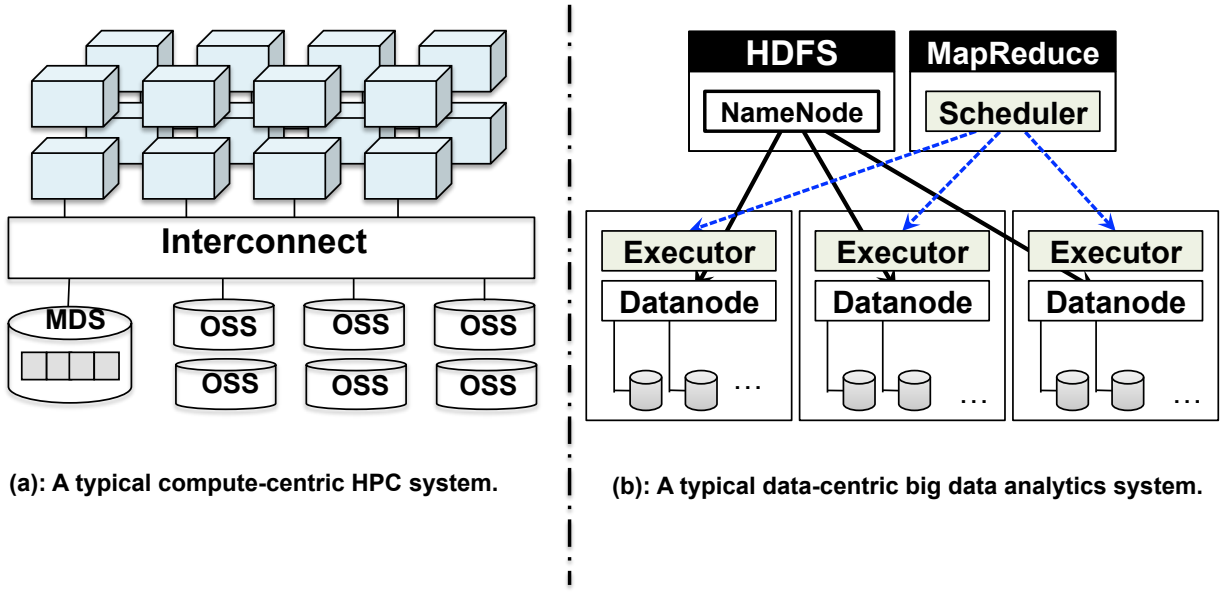


Figure 5.1: Detailed comparison between compute- and data-centric paradigms.

Figure 5.1(a) shows a diagram of typical compute-centric HPC systems. The core of such systems consists of a large collection of compute nodes, *i.e.,* processing elements (PEs), which offer the bulk of computing power. Via a high-speed interconnect, these PEs are connected to a parallel file system from the storage backend for data I/O. Lustre is a typical file system used on HPC systems. It is a POSIX-compliant, object-based parallel file system, offering parallel I/O services to the clients (PEs) through a MetaData Server (MDS) and many Object Storage Servers (OSSes).

Lustre provides fine-grained parallel file services with its distributed lock management. To guarantee file consistency, it serializes data accesses to a file or file extents using a distributed lock management mechanism. Because of the need for maintaining file consistency, all processes

first have to acquire locks before they can update a shared file or an overlapped file block. Thus, when all processes are accessing the same file, their I/O performance is dependent not only on the aggregated physical bandwidth from the storage devices, but on the amount of lock contention among them as well.

5.3 Methodology

5.3.1 Experimental Testbed

Table 5.1: List of key Spark configuration parameters.

Parameter Name	Value
spark.reducer.maxMbInFlight	1GB
spark.rdd.compress	false
spark.shuffle.compress	true
spark.buffer.size	8MB
spark.default.parallelism	application dependent

Unless otherwise specified, our experiments are carried out on the Hyperion cluster [11] with 101 compute nodes at Lawrence Livermore National Laboratory. One node serves as the master of the Spark and the NameNode of HDFS. Each compute node is equipped with two 2.60*GHz* Intel E5-2670 processors (16 cores per node) and 64 GB of RAM. We allocate 30 GB per node for Spark jobs and reserve 32 GB for RAMDisk. On each node, there is one SATA-based SSD of 128 GB storage space mounted via *ext4* file system. Its peak sequential write and read bandwidths reach 387 MB/sec and 507 MB/sec, respectively. All compute nodes span across two racks and are fully connected through InfiniBand QDR which delivers up to 32 Gpbs link bandwidth. A centralized Lustre file system providing 47 GB/sec aggregated bandwidth is mounted on all the compute nodes.

All compute nodes run Linux 2.6.32 kernels. Spark 0.7.0, along with Scala 2.9.2 and Oracle Java 1.7.0 are used. The HDFS block size is set as 128 MB. We have also carefully tuned Spark on Hyperion. Table 5.1 summarizes main parameters that have noticeable performance impact. For all tests, we report the median of five test runs.

5.3.2 Benchmarks

We have selected three representative benchmarks including *GroupBy*, *Grep*, and *Logistic Regression* (LR). They are described as follows.

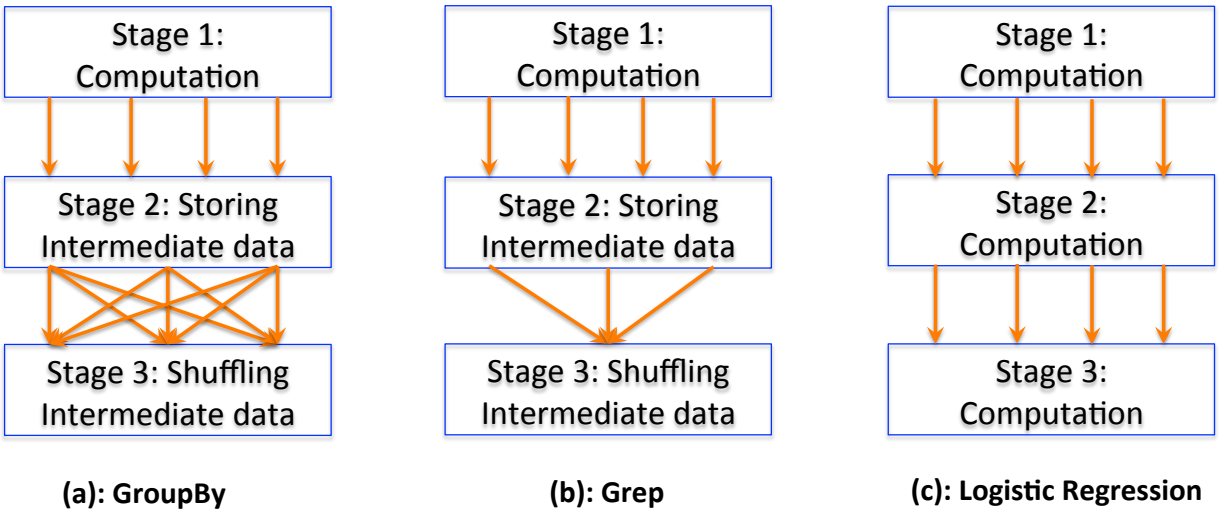


Figure 5.2: Execution plans of three representative benchmarks.

GroupBy is a critical operation used by many applications, including *kMeans*, *wordcount*, and *calculating transitive closure of a graph*, *etc*. It helps reveal the pattern of shuffle operations. Figure 5.2(a) depicts the execution plan of GroupBy. It consists of three stages. In the first computation stage, each task generates <key, value> pairs in memory. In the second stage, Spark schedules ShuffleMapTasks to partition the intermediate data and store them into the file systems. In the last stage, fetching tasks shuffle intermediate data over the network. Across such data processing pipeline, the intermediate data size is equal to the input size.

Grep searches a string that matches a regular expression from a set of documents. It represents a wide range of data analytics applications, such as *logQuery* and *select*, *etc*. Grep's execution plan as shown in Figure 5.2(b) bears some similarity to that of GroupBy. However, it generates much less intermediate data, requiring very little shuffling of data. Its intermediate data size ranges from 1 MB to 200 MB in our test cases.

***Logistic Regression* (LR)** is an iterative application that predicts the value of a vector according to a qualitative response model. It can leverage the strength of Spark in caching job results in memory. As shown in Figure 5.2(c), we run three iterations for LR. Every iteration is translated into one Spark job that is executed in one stage. Multiple stages are not pipelined in this benchmark.

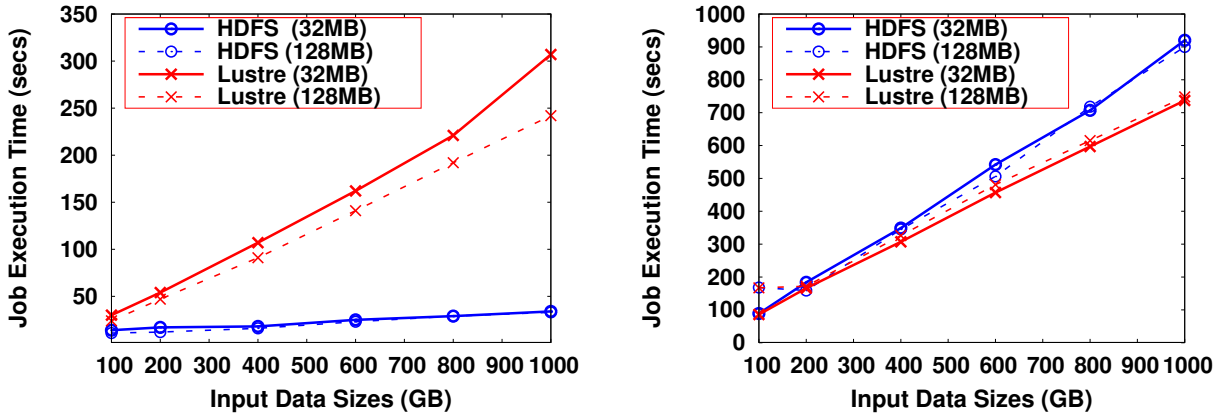
5.4 The Impact of Storage Architecture

As discussed in the introduction, the storage architecture is a key distinction between data- and compute-centric paradigms. In the data-centric paradigm computation tasks are co-located with the storage resources, while in compute-centric paradigm tasks need to access a separate storage subsystem via interconnect. In this section, we characterize the impact of storage architecture on MapReduce jobs. To have a storage architecture for the data-centric paradigm, we configure an HDFS file system with 32 GB RAMDisk as the storage for each DataNode on Hyperion. For the storage architecture of the compute-centric paradigm, we directly use the Lustre file system of Hyperion.

5.4.1 Location of Data Source

Among the three benchmarks, both Grep and LR work with a varying amount of input data. But they differ significantly in terms of their analytics computation. Grep generates a small amount of intermediate data, for which shuffling is required; and LR does mostly computation. We run both benchmarks with their input coming from the compute-centric *Lustre*-based configuration and the data-centric *HDFS*-based configuration.

Figure 5.3 shows the comparison of the job execution time of Grep and LR benchmarks for both configurations. Overall, we have observed that the extent of impact is highly dependent on the computational intensity of MapReduce tasks. For Grep jobs with low computation, such as simply scanning of the input, the Lustre configuration results in severe performance penalty. Figure 5.3(a) shows that, with 32 MB split size, the compute-centric Lustre configuration performs up to 5.7×



(a) Grep.

(b) Logistic Regression (LR).

Figure 5.3: Performance of retrieving inputs from HDFS and Lustre.

worse than HDFS on average. For the Lustre configuration, increasing the split size from 32 MB to 128 MB reduces the job execution time by 15.9% due to less scheduling overhead. But there is still a significant performance loss when running Grep on the compute-centric Lustre configuration.

On the contrary, for the computation-intensive jobs, such as multidimensional vector multiplication in LR, the cost of retrieving input from Lustre is not as significant as shown in Figure 5.3(b). Furthermore, as shown in the figure, the Lustre configuration outperforms HDFS by 12.7% on average for a 32 MB split size. This improvement is consistent across different split sizes. The performance difference is caused by delay scheduling policy [102] adopted by Spark, which will be further analyzed in Section 5.5.1.

Taken together, the impact of the storage architecture to MapReduce applications depends on the characteristic of the applications' computation tasks. For LR-type computation-intensive jobs, the impact is negligible. But for Grep-based jobs with low computation requirements, the compute-centric Lustre configuration negatively affects the performance.

5.4.2 Location of Intermediate Data

The location of intermediate data is another critical issue. It directly determines the performance of intermediate data shuffling. To investigate this factor, we use the GroupBy benchmark that allows flexible tuning of the intermediate data size. During the evaluation, we run GroupBy and store the intermediate data to the two different storage configurations.

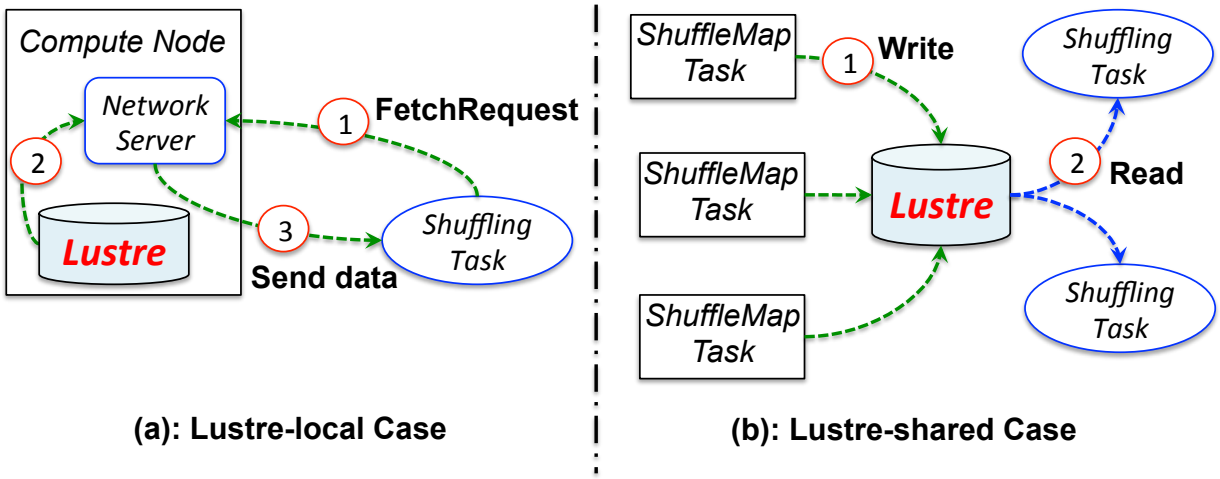


Figure 5.4: Two approaches to use Lustre to conduct intermediate data shuffling.

Fig. 5.5(a) illustrates the performance of GroupBy when intermediate data resides in different storage architectures. Overall, the data-centric HDFS configuration exhibits significant advantage over the compute-centric alternative. It outperforms the optimal Lustre case (Lustre-local) by up to 6.5× on average, and the improvement ratio increases linearly with the size of intermediate data. However, due to the limited storage spaces, HDFS can only support a maximum of 1.2 TB intermediate data size.

However, in many scenarios, compute nodes in HPC clusters are not equipped with any local persistent storage systems, for which placing intermediate data on the compute-centric Lustre-based storage is the only choice.

Lustre-local and *Lustre-shared*, as shown in Fig. 5.4, illustrate two approaches to use Lustre for intermediate data shuffling. In the Lustre-local case, fetching tasks that need to shuffle the intermediate data are unaware of the existence of the Lustre. Thus they initiate FetchRequests to

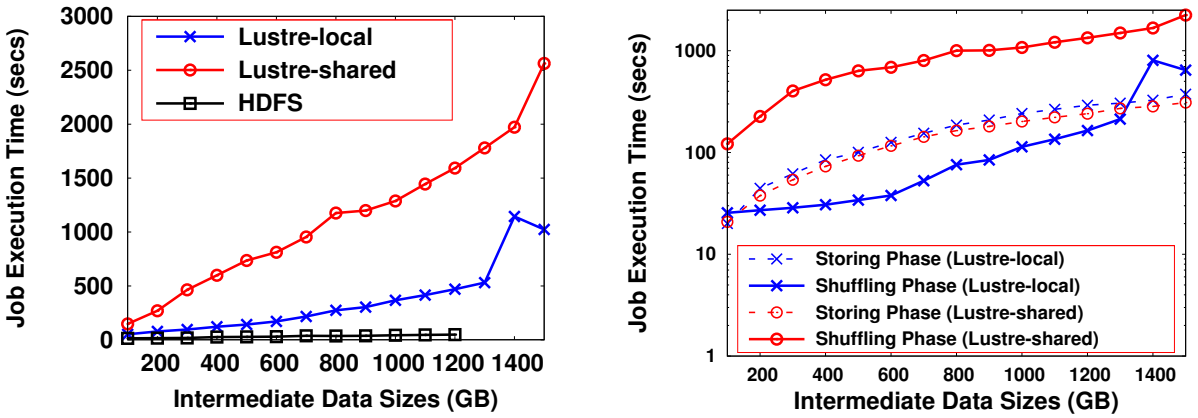
the remote servers which in turn retrieve the data from their local Lustre directories and send them back across the network. However, since data retrieval from Lustre requires a movement over the network, Lustre-local can cause repetitive data movements, wasting the network bandwidth.

We have examined the alternative Lustre-shared approach, in which each fetching task directly retrieves intermediate data from Lustre. Although this approach seemingly addresses the issue of repetitive data movement within Lustre-local, it suffers from tremendous performance degradation due to file consistency ensured by Lustre.

Fig. 5.5(a) illustrates that Lustre-shared performs worse than Lustre-local by up to 3.8× with GroupBy benchmark. The detailed dissection in Fig. 5.5(b) further reveals that, although the two approaches perform comparably in the data storing phase, the shuffling phase of Lustre-shared is inferior to that of Lustre-local by up to one order of magnitude. The main reason for the inferiority of Lustre-shared is that retrieving intermediate data written by remote servers incurs costly metadata operations at the OSSes due to the need to maintain the storage consistency.

In the Lustre-local approach, the server that handles the FetchRequests simply retrieves the intermediate data written by the tasks on the same node. Meanwhile, due to the effect of large buffer cache in a compute node, it is likely that those intermediate data and corresponding metadata, such as *write locks*, still reside in the local memory. Thus, they can be quickly retrieved to serve the FetchRequests without involving expensive internal operations of Lustre for maintaining the data consistency.

On the contrary, in the Lustre-shared case, each fetching task accesses Lustre to retrieve the data written by remote nodes. Such design requires the *Distributed Lock Manager* of Lustre to revoke the write locks. After lock revocation, intermediate data cached remotely is forced to be flushed to the OSSes before they become available to fetching tasks. This sequence of internal operations substantially delays the intermediate data movement. Furthermore, current Spark launches fetching tasks of a job simultaneously during the shuffling phase, forcing all the intermediate data to be flushed to the OSSes around the same time. As a result, such behavior can cause serious contention at Lustre, significantly degrading the performance of the shuffling phase.



(a) Job Execution Time of GroupBy.

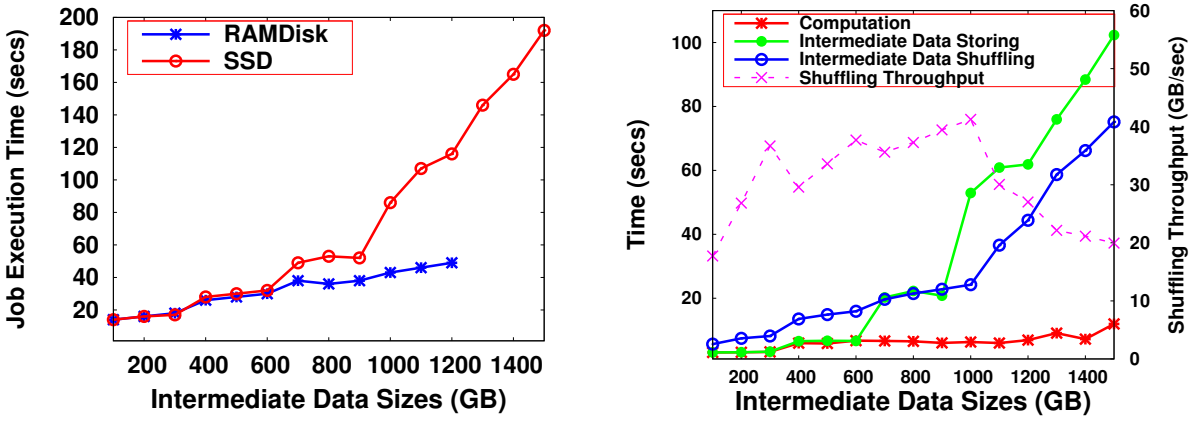
(b) Dissection of Lustre Cases.

Figure 5.5: Performance when intermediate data resides in Lustre.

In summary, the data-centric HDFS configuration shows dramatic advantage over the compute-centric configuration when used for storing the intermediate data. In the compute-centric case with a shared file system such as Lustre, fetching tasks can avoid costly metadata operation for better performance if they are oblivious to the features of the shared file system.

5.4.3 Leveraging Solid State Disks

Many HPC systems are embracing a hierarchical stack of different storage devices in order to support both data-centric and compute-centric paradigms, so that they can support both traditional HPC applications and emerging data analytics programs. A major effort to achieve such goal is the trend to integrate high-performance Solid State Drives (SSD) to the compute nodes. An immediate impact to MapReduce is that they can efficiently facilitate the processing of intermediate data. To understand such performance implication, we have conducted a set of experiments to study the performance impact of SSD on MapReduce jobs with similar data-centric HDFS configuration as that in Section 5.4.2. The performance of using RAMDisk as the local persistent storage is employed for performance comparison. We continue to use GroupBy as the benchmark for this study.



(a) Job Execution Time.

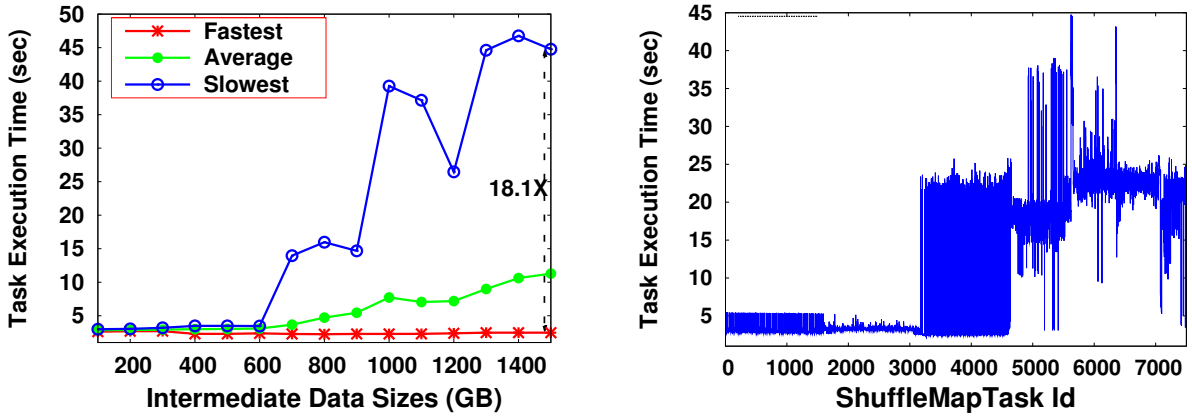
(b) Detailed dissection.

Figure 5.6: Performance when SSD is used for storing the intermediate data.

Figure 5.6(a) presents the job execution time of GroupBy when intermediate data is stored on RAMDisk and SDD, respectively. Overall, using SSD for intermediate data achieves comparable performance as RAMDisk when the data size ranges from 100 GB to 600 GB due to the caching effects from the file system. Once the data size exceeds 700 GB, RAMDisk performs substantially better than SSD. Note that SSD can support jobs with much larger intermediate data sizes than RAMDisk due to the capacity advantage of SSD.

Figure 5.6(b) further shows a detailed dissection of job execution time when SSD is employed. Data shuffling is shown as the key bottleneck when data size $\leq$ 600 GB, in which the throughput is bounded by the network bandwidth. When the data size is between 700 GB and 900 GB, the cache can no longer satisfy all the write operations during the storing phase. As a result, both storing and shuffling of intermediate data contribute equally to the job execution. When the data size increases further beyond 900 GB, we observe sharp drops on the performance of storing and shuffling phases due to the degraded performance of SSD write and read operations. In addition, the write performance falls more drastically than that of read. When the storing phase of intermediate data becomes the major bottleneck of job execution, the throughput of data shuffling then becomes SSD-bound.

5.4.4 Inefficiency in Utilizing SSD



(a) Performance variation among tasks that write SSDs.

(b) Execution time of all ShuffleMapTasks.

Figure 5.7: Detailed analysis of tasks that write to and shuffle data from SSDs.

In our experiments with SSD, there is a significant performance variation among *ShuffleMapTasks* writing intermediate data to SSDs as shown in Figure 5.7(a). The performance gap between the fastest and the slowest tasks can be as wide as 18× when the data size reaches 1.5 TB. On the contrary, the performance variation among shuffling tasks is moderate (not shown for brevity), indicating a mild interference among SSD read operations.

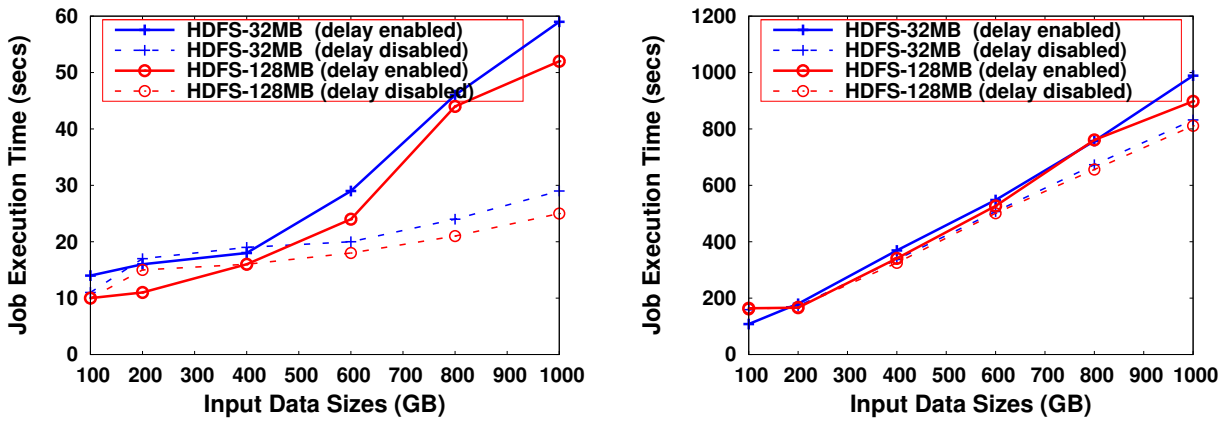
The dramatic variations among *ShuffleMapTasks* is because Spark aggressively launches tasks as they arrive in order to reduce the latency. This is oblivious to the congestion of underlying SSDs. When multiple data-intensive tasks are running and issuing a large number of write requests, such oblivity can result in substantial interference amony tasks. To gain insight into this issue, we have profiled the execution times of all ShuffleMapTasks in the 1.5 TB test case. We plot the execution times of these tasks based on the order of their launch time in Figure5.7(b). As shown in the figure, early tasks can take advantage of write buffer and clean blocks on SSDs. They can quickly complete their work. When the buffer gradually fills up and clean SSD blocks are depleted, internal operations for delayed write and garbage collection are activated. These operations start

to interfere with the execution of ShuffleMapTasks. Thus we observe a degraded performance for Tasks ranging from 3100 to 4500. However, Spark is unaware of such interference and continues to insert tasks. This behavior further exacerbates the contention on the SSDs and leads to severer interference among Tasks from 4800 to 6400.

In summary, our study reveals that the lack of awareness on the unique features of SSD can lead to inefficient utilization of resource when in the storage of intermediate data. Fortunately, the inefficiency of congestion-oblivious write has also been documented by many prior studies on SSD [4, 28, 77]. In Section 5.6.2, we will demonstrate that an optimization using a throttling mechanism can effectively mitigate the interference and improve the storing phase by 41.2%.

5.5 The Impact of Data Locality and Task Scheduling

5.5.1 Locality-Oriented Scheduling



(a) Grep.

(b) Logistic Regression (LR).

Figure 5.8: Performance degradation caused by delay scheduling.

Maximizing data locality has been a critical objective of MapReduce schedulers [102, 88, 91, 67]. *Delay scheduling* [102], adopted by Spark, is a notable effort for obtaining high data locality for MapReduce frameworks in the environments where network bandwidth is a scarce resource.

Using the same compute- and data-centric configurations as described in Section 5.4, we conduct an experiment to characterize the importance of locality-oriented scheduling.

Figure 5.8 shows the experiment results when we activate delay scheduling for the data-centric HDFS configuration. When the split size is equal to 32 MB, job execution time degrades by 42.7% and 9.9% on average for Grep and LR, respectively. Similar degradation occurs for other split sizes as well. In contrast, with the compute-centric Lustre configuration, tasks can be immediately launched on available compute nodes since there is no locality constraint. All the computation tasks are roughly at the same distance from storage resources. Thus, compared to the data-centric configuration that favors the use of delay scheduling for better data locality of tasks, this setting can benefit the computation-intensive MapReduce jobs as shown in Figure 5.3(b),

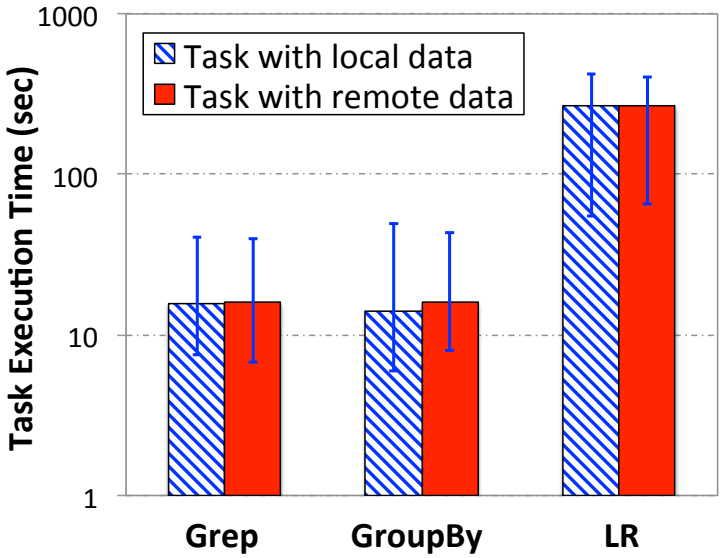


Figure 5.9: Task execution time of three benchmarks.

In addition, Spark pipelines computation with data input, further diminishing any benefit of data locality. Figure 5.9 demonstrates such argument. It shows the comparison of average task execution times along with maximum and minimum values of three different benchmarks. *"Task with local data"* denotes that the data input is obtained locally, while *"Task with remote data"* indicates the data input from remote servers. As shown in the figure, enforcing tasks to achieve 100% locality provides little performance gain for all three benchmarks.

Taken together, our evaluation and characterization of locality-oriented scheduling for the compute- and data-centric configurations suggest that (1) scheduling for good locality may not be effective in improving the performance of MapReduce jobs in HPC environments, and (2) introducing delays for better task locality is even detrimental on compute-centric systems because of the uniform reachability of storage resources to all computation tasks.

5.5.2 Load Balance of MapReduce Tasks

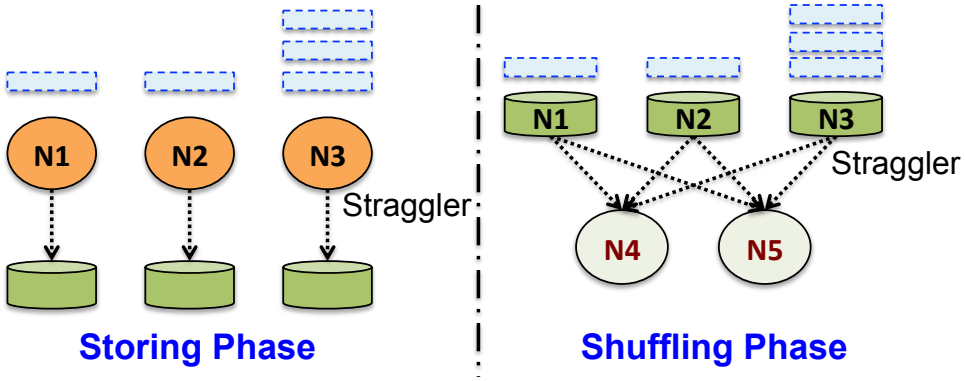
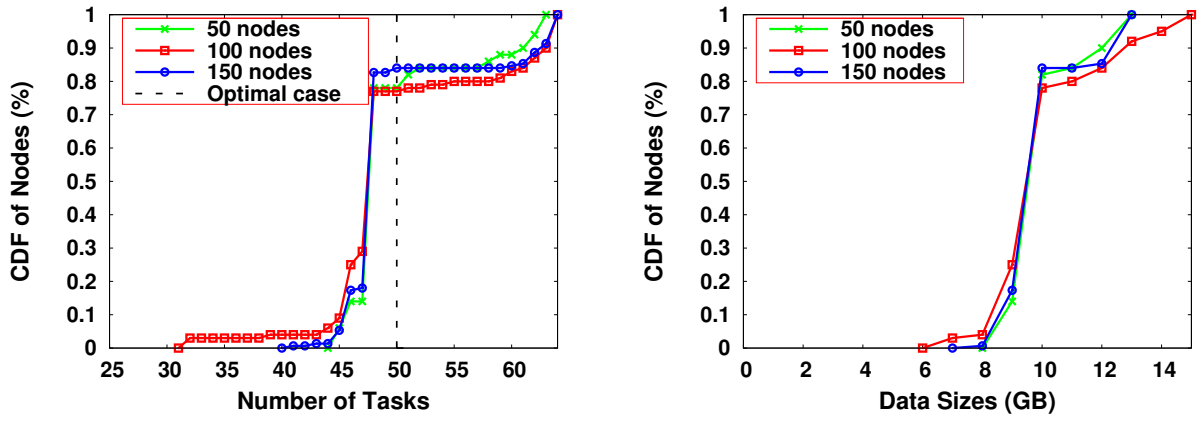


Figure 5.10: Straggler issue caused by imbalanced intermediate data distribution during I/O intensive shuffle operation.

Although the compute nodes in a compute-centric environment are homogeneous, there exist performance variations among compute nodes due to the skew of workloads over time. As a result, fast nodes tend to be assigned with more tasks by the scheduler. When each of these tasks deposits a unit of intermediate data, fast nodes end up with much more data to shuffle or move. This leads to imbalanced distribution of intermediate data. When a shuffle operation is needed, such imbalanced distribution can cause straggler issue [25] that prolongs the ensuing I/O-intensive data storing and shuffling phases as depicted in Figure 5.10.

To investigate this issue, we use GroupBy as the benchmark with a split size of 256 MB. Three sets of experiments are conducted to run 2500 tasks on 50 nodes, 5000 tasks on 100 nodes, and 7500 tasks on 150 nodes, respectively. Figure 5.11 (a) and (b) illustrate the *cumulative distribution functions* (CDF) of task and intermediate data distributions.



(a) Task distribution.

(b) Intermediate data distribution.

Figure 5.11: Unbalanced task assignment leads to unbalanced intermediate data distribution.

As shown in Figure 5.11 (a), the workload among compute nodes varies substantially. In the case of 100 nodes, for the first 3% nodes at the head of the distribution, each machine only hosts 7 GB of intermediate data. While for the last 10% nodes at the tail of the distribution, each node accommodates more than 14 GB, i.e., 2× of workload difference. Because the execution time of storing and shuffling phases are directly determined by the slowest tasks, those nodes with the most intermediate data can severely drag down the performance regardless of how fast other tasks have achieved.

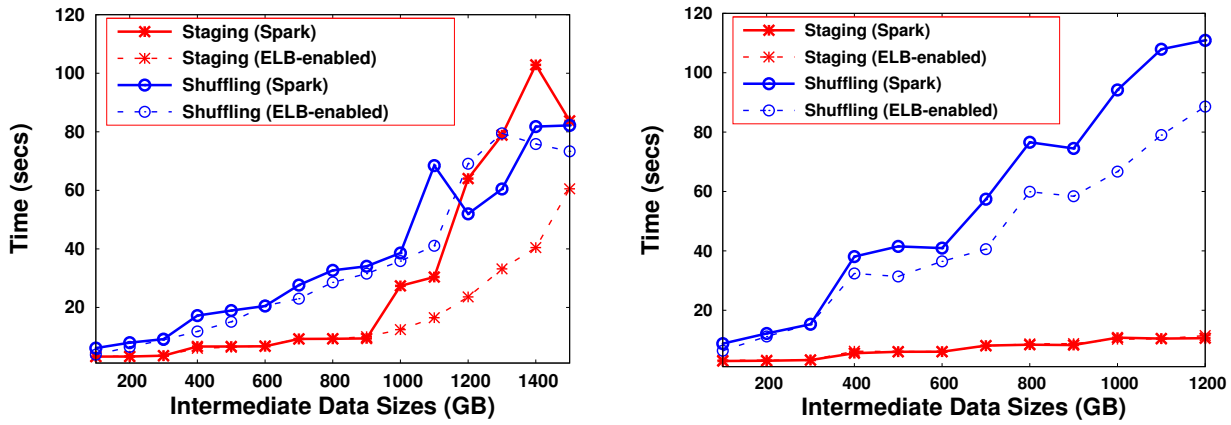
In summary, performance variations and workload skews on compute-centric systems can lead to imbalanced distribution of both MapReduce tasks and their intermediate data. Without an appropriate solution, such issue can hinder MapReduce systems from achieving the best performance on compute-centric HPC systems. We will demonstrate in Section 5.6.1 that, by taking into account of the intermediate data size, the shuffle operation can be effectively accelerated.

5.6 Optimizations for Spark on Compute-Centric HPC Systems

Based on the characterization from Sections 5.4 and 5.5, we have shown that there are two performance issues that need to be addressed for the memory-resident Spark framework in order

for it to be effectively supported by the compute-centric HPC systems. Firstly, the scheduler should take into account of the need to balance the intermediate data among compute nodes and mitigate the variations of task execution, thereby avoiding stragglers. Secondly, the MapReduce workers should be aware of the unique features of hierarchical storage devices such as SSDs to effectively utilize them. Accordingly, we introduce two optimizations: namely *Enhanced Load Balancer* (ELB) and *Congestion-Aware task Dispatching* (CAD), to address these issues.

5.6.1 Enhanced Load Balancer (ELB)



(a) Storage is the bottleneck.

(b) Network is the bottleneck.

Figure 5.12: Dissection of GroupBy job execution time.

We design ELB to address the issue of imbalanced distribution of intermediate data. It considers the size of intermediate data generated by tasks before making further task assignment decision. When a job starts, ELB-enabled scheduler assigns tasks to the workers in a round-robin manner. During the job execution, ELB records the amount of intermediate data generated by each completed task and monitors the average data size among all nodes. When the size on a node goes beyond the average by a threshold (25% currently), ELB notifies the scheduler to stop assigning more tasks to that worker node. Instead, it picks the nodes hosting the least amount of intermediate

data to execute the pending tasks. Once the average size goes up, ELB resumes to assign more tasks to the original heavily loaded worker.

Although ELB can balance the size of intermediate data among compute nodes, two issues arise under such design. Firstly, ELB may conflict with the data locality, since the nodes hosting the least amount of intermediate data may not possess the input for the tasks. However, as shown in Section 5.5.1, enforcing data locality has negligible impacts on the task execution time in the HPC environment. Thus it is desirable to trade off the locality of task scheduling for a balance of data distribution. Secondly, ELB can cause the idling of certain workers when they have completed their share of computation tasks, and the entire computation phase can be consequently delayed due to the slowest task. However, we have observed that the cost of waiting for the slowest computation task is much less than the cost of waiting for the slowest I/O tasks.

In this context, to demonstrate the performance improvement of ELB-enabled scheduler to communication and storage bottlenecks during the shuffle operations, the GroupBy benchmark is used. To create a scenario of storage bottlenecks, SSD is used as the local storage device. In Hyperion, we are not allowed to use other networks other than InfiniBand. So to create a scenario of network bottlenecks, we reduce the data size set in FetchRequest from 1 GB to 128 KB. Thus, many more requests are needed to shuffle the same amount of data, and the network bandwidth is consequently narrowed (due to space constraint, we only present the dissection of job execution and omit execution time of computation phases for clearness).

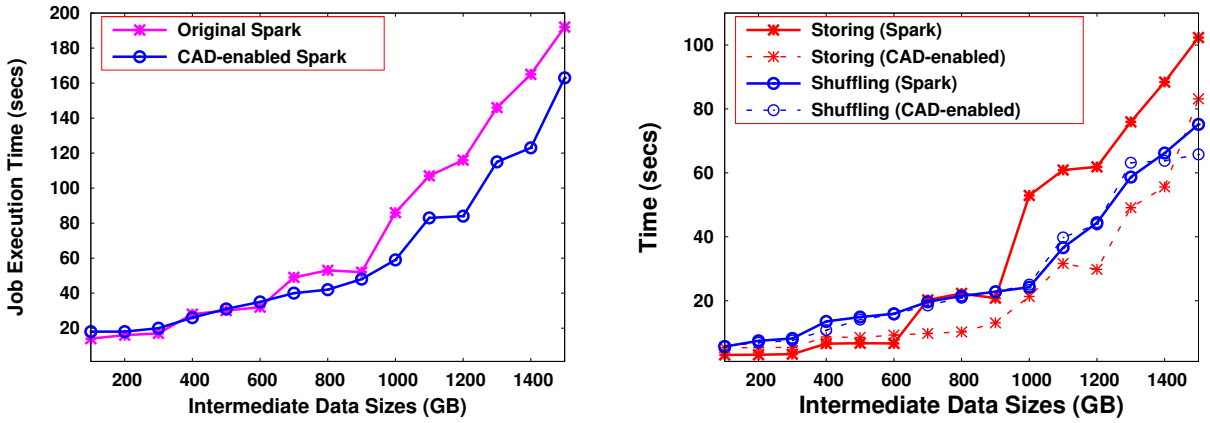
Storage Bottlenecks: Figure 5.12(a) shows that when the data size $\leq$ 900 GB, Spark and ELB perform similarly. However, ELB outperforms Spark by 26% on average in terms of job execution time when the data size is between 1 TB and 1.5 TB. Such improvement is mainly attributed to the accelerated staging phase introduced by the ELB. When data sizes reach beyond 1 TB, Spark performs worse than ELB by 2.2$\times$ on average in the staging phase. On the contrary, computation phases from both remain nearly the same.

Network Bottlenecks: Spark performs 14.8% worse than ELB on average in terms of job execution time. Moreover, when the network is the bottleneck, unbalanced distribution has severe

impact on small datasets, showing up to 17.5% degradation when data size is 400 GB. Such difference is strongly determined by the shuffling phases as shown in Figure 5.12(b). On average, Spark shuffles data slower than ELB by 29.1% when the input size ranges from 400 GB to 1.2 TB.

Taken together, our ELB demonstrates that unbalanced distribution of intermediate data can prevent memory-resident Spark from achieving the optimal performance in the HPC environment.

5.6.2 Congestion-Aware Dispatching (CAD) of Tasks



(a) Job execution time.

(b) Dissected of job execution.

Figure 5.13: Performance of Congestion-Aware task Dispatching.

We design CAD as a feedback control algorithm that aims to mitigate the task interference when SSD is used as the storage device for intermediate data. CAD speculates the congestion status of SSD devices by monitoring the task execution time of completed ShuffleMapTasks. When a significant jump of execution time is detected, it throttles the dispatching of tasks by introducing a delay interval before each dispatching step. In the current design, we increase the interval by 50 ms whenever the average execution time increases by 2× (these are empirically chosen during our tuning process). Conversely, we reduce the interval accordingly when the average task execution time drops by half. Though simple, we have observed that such mechanism is effective in optimizing the SSD writes. This is because such delay interval allows more time for outstanding operations

inside SSD to complete their work without worsening the congestion. In addition, it also provides more opportunity to group many small writes, which are harmful to SSD, thus further reducing the interference.

Figure 5.13 compares the performance of the original Spark with our CAD-enabled Spark by using the GroupBy benchmark with different input sizes. Overall, CAD effectively accelerates the intermediate data storing phase once the data size goes beyond 600 GB. It achieves this without affecting another two phases as shown in Figure 5.13(b). On average, CAD reduces the storing phase by up to 41.2% when the data size ranges from 700 GB to 1.5 TB. Such acceleration is reflected in the job execution time as shown in Figure 5.13(a). The average improvement ratio reaches 19.8%.

5.7 Discussion

In this section, we summarize our major findings and discuss their implications to the design of future systems.

The Impact of Storage Architecture: Computation intensity of MapReduce tasks determines how much impact the storage architecture of HPC systems will have on the job execution. For computation-intensive applications, there is little impact between the storage architectures of data- and compute-centric paradigms. In addition, there is no locality to the storage for compute nodes on compute-centric systems; tasks can be launched on any node with little loss of performance, or even better performance compared to the data-centric environment. However the data-centric paradigm still exhibits superior performance for applications with low computation intensity and high data intensity. Therefore, it is critical to consider the characteristics of MapReduce jobs before making data placement decisions. This is important for system providers in planning the evolvement of their compute-centric HPC systems for data-centric analytics applications.

In addition, the storage architecture may use distributed locking mechanism for maintaining file consistency, which can severely degrade the performance of intermediate data movement. So we show that designing shuffling mechanisms can avoid the cascading effects of locking contention

and keep the efficiency of intermediate data shuffling. Users need to avoid a pitfall to use traditional HPC parallel file system as a bridge for fast storage of intermediate data.

When SSDs are used as the storage device for intermediate data, our analysis shows that Spark is currently incapable of utilizing them efficiently. Uncoordinated resource utilization can cause severe congestion on the device, leading to significant task interference, as also shown in [59]. Our findings suggest that comprehensive examinations are needed to assure the performance of MapReduce applications while evolving the underlying storage of a system to SSDs. Optimization strategies, such as *task throttling* as shown by our study, can be leveraged to improve the efficiency of SSD device utilization.

The Effectiveness of Locality-Oriented MapReduce Schedulers in HPC Environment: Our characterization reveals that, when a data-centric storage architecture is configured for computer nodes of an HPC system, MapReduce schedulers that strive for maximum data locality is not critical. Moreover, they may even hurt the performance by forcing a task delay for future opportunistic locality. We have also revealed that while HPC systems generally have homogeneous computer nodes, load imbalance can still arise. The current scheduler is oblivious to the size of intermediate data generated by computation tasks, leading to imbalanced data distribution that can cause many stragglers during shuffle operations. Our study demonstrates that such imbalanced distribution can cause suboptimal performance to MapReduce jobs. MapReduce applications on HPC systems shall not focus on locality-oriented task scheduling but other critical factors such as balancing distribution of intermediate data.

5.8 Related Work

Spark is a critical cornerstone of *Berkeley Data Analytics Stack* (BDAS) [5] that aims to compete with the open-source Hadoop. It plays a pivotal role in many industry and academia projects [93, 105, 104, 20, 58] *etc.* Shark [93] is a query processing framework on top of Spark. It compiles user-submitted SQL queries into Spark jobs and leverages optimization strategies commonly used in database systems to optimize the execution plan. Also coupled with Spark, BlinkDB [20]

is another approximate query engine that trades query accuracy for response time so that it can delivery near instant response for interactive queries over massive scale datasets. Spark streaming [105, 104] exploits the potential of Spark to process real-time streaming data. It partitions streaming computations into small-sized deterministic batch jobs to fit the computation model of Spark. Sparkler [58] optimizes the Spark to support large-scale matrix factorization more efficiently. It identifies a major inefficiency existing in current Spark's broadcast variable and introduces a *Carousel Maps* to spread large dataset via using distributed hash table. Our work is orthogonal to those efforts. In addition, Zaharia *et al.* have introduced LATE [106]. Ananthanarayanan *et al.* have introduced Mantri [25] and small job cloning [23] to mitigate the impact of stragglers. However, none of them considers the imbalanced intermediate data distribution issue.

Many parties have tried to incorporate MapReduce frameworks with distributed file systems for compute-centric paradigm. Ananthanarayanan *et al.* [26] evaluated MapReduce when it runs with HDFS and GPFS. Maltzahn *et al.* [27] studied the combination of Hadoop with Ceph file system. Panasas [1] is also delivering the support for Hadoop. Our analysis in this work provides researchers with the first hand data about absorbing MapReduce into compute-centric HPC paradigm that relies on above high-performance file systems.

Many efforts have been conducted to investigate the performance of HPC applications on data-centric cloud. Evangelinos *et al.,* [34] analyzed a scientific HPC application on Amazon EC2 and revealed that the performance of network in cloud is worse than that of HPC by one to two orders of magnitude. Gupta *et al.,* [39] observed similar performance on different cloud platforms. Though raw performance difference between compute-centric HPC and data-centric cloud is pronounced, Marathe *et al.,* [63] pointed out that queue wait time is another critical factor to consider when choosing which environment is the best for the applications. Our work stands on the other side of the spectrum by investigating the data-centric analytics framework on compute-centric paradigm.

5.9 Summary

While many existing HPC facilities are evolving new capabilities to support efficient analytics of big data, the research in this chapter addresses an important question on how to support the traditional compute-centric paradigm for HPC applications and the emerging data-centric paradigm for big data analytics applications on the same HPC systems. We have examined the design and architecture of a state-of-the-art MapReduce framework – Spark – on HPC systems. Our work sheds light on the performance issues and design inefficiency when running Spark jobs on HPC systems with distinct data-centric and compute-centric configurations. In particular, we have investigated the impact of storage architecture, locality-oriented scheduling, and emerging storage devices to memory-resident MapReduce applications on HPC systems. Based on the experimental results, our optimization techniques, including the Enhanced Load Balancer and the Congestion-Aware Task Dispatching, can efficiently improve the performance of Spark applications on HPC systems.

Chapter 6

Conclusion

To this end, this dissertation stands as a substantial effort to optimize contemporary MapReduce systems from three dimensions simultaneously, including optimizing single job execution, provisioning both fairness and efficiency, as well as enhancing HPC cluster utilization. Meanwhile, it has holistically examined, quantified and elucidated the limitations, bottlenecks, and design inefficiencies in major MapReduce frameworks, including Hadoop and Spark. Based on the comprehensive analysis results, this dissertation has made following three key contributions to better prepare next-generation MapReduce systems for addressing the big data challenge.

Network Levitated Merge Algorithm: This dissertation has provided MapReduce systems with high-performance I/O services to efficiently accelerate their intermediate data movement. It introduces a novel Network Levitated Merge algorithm in Chapter 3, along with a Hadoop Acceleration framework that together overcome three critical performance issues, including a serialization barrier between shuffle/merge and reduce phases, repetitive merge within the ReduceTasks, and lack of capability to leverage fast networks. By completely merging intermediate data in memory and supporting high-performance network protocol to transfer data, Network Levitated Merge algorithm significantly improves job execution time, reduces the CPU utilization, meanwhile achieving comparable scalability as original Hadoop MapReduce. Furthermore, Hadoop Acceleration project has also laid down a solid foundation for future researches on improving the I/O performance of MapReduce. The publications from this project include [90, 89, 96, 59, 68, 100, 99]

Preemptive ReduceTask based Fast Completion Scheduler: This dissertation has revealed that current MapReduce schedulers are insufficient to deliver ideal quality-of-service in a multi-tenant MapReduce clusters. In particular, this dissertation has identified and quantified the overhead imposed by the job starvation and resource underutilization problems raised by non-preemptive

long running ReduceTasks. In addition, it has also recognized the fundamental inefficiency that contemporary schedulers treat ReduceTasks similarly to MapTasks without regard to their distinct execution patterns, in which ReduceTasks aim to attain as much I/O efficiency as possible while MapTasks aim to obtain the fastest computation. Consequently, MapReduce schedulers can significantly penalize small jobs under concurrent workloads, causing severe unfairness towards small jobs in the multi-tenant MapReduce clusters. Accordingly, this dissertation has introduced Preemptive ReduceTask and Fast Completion Scheduler in the Chapter 4 to address those issues. Preemptive ReduceTask is a lightweight work-conserving preemption mechanism that allows ReduceTasks to be preempted at anytime during their execution without losing accomplished I/O and computation work. Therefore, it enables flexible task execution control to prioritize short ReduceTasks from small jobs. Leveraging the advantages of Preemptive ReduceTask, this dissertation has further introduced Fast Completion Scheduler that considers both efficiency and fairness when scheduling concurrently running jobs. Our experimental evaluation with a diverse collection of workloads adequately demonstrate that Fast Completion Scheduler substantially outperforms the state-of-the-art scheduling policies in terms of quality-of-services.

Enhancing the Adaptability of MapReduce on HPC Platforms: MapReduce has been regarded by scientists in leadership computing facilities as a promising solutions to process gigantic simulation results. However, there is a lack of research to study the performance characteristics of MapReduce systems on HPC platforms. This dissertation has filled this void by thoroughly analyzing the performance impact of different storage architectures, consistency guarantees provisioned by parallel file systems, storage devices and locality-oriented scheduling on various of MapReduce jobs in the Chapter 5. Our evaluation results suggest that even the state-of-the-art MapReduce system is unable to exploit the optimal performance from the HPC systems. Accordingly, this dissertation has introduced two optimization solutions, named as Enhanced Load Balancer and Congestion-Aware task Dispatching, to enhance the adaptability of MapReduce for HPC platforms. Our performance investigation further demonstrates that with the equipment of above two enhancements, Spark can achieve much higher throughput than the otherwise.

Chapter 7

Future Work

This dissertation has also opened up many opportunities for future MapReduce research. Particularly, the following three future studies.

Enhancing Failure Recovery: Although MapReduce frameworks are designed as fault-tolerant systems, little research has been carried out to comprehensively investigate the performance of MapReduce under various failure scenarios. While, our experience of dealing with different types of failures during studying the aforementioned techniques indicates that significant performance penalty can be imposed on job execution time when failures occurs.

However, conventional wisdoms in designing contemporary MapReduce frameworks simply employ task re-execution with several tries regardless of the root causes of the failures. When tasks exhibit long-running, multi-phases execution patterns, such solution can severely degrade the performance of MapReduce programs. Although several studies [69, 54] have attempted to leverage multi-replicas checkpointing to persist the intermediate data into distributed file system so as to avoid task re-execution when failures occur, they are at the high expense of much longer normal task execution time, thereby not ideal solutions.

Our initial studies show that lightweight preemption mechanism and pipelined ReduceTask execution introduced in Chapter 4 and Chapter 3 respectively demonstrate effective performance results when leveraged for facilitating MapReduce recovery from failures. Since it is critical to improve failure resilience, we will pursue this directions with the techniques we have mastered in this dissertation.

Lightweight Task Migration: Task Migration has been introduced in Chapter 4. However, detailed performance investigation is still an on-going work. Contemporary MapReduce systems are not supporting efficiently lightweight task migration. To move a task from one node to the

other, killing the task then re-launching it is the only option with the need to repeat the previously accomplished I/O and computation work. Such strategy can impose heavy overhead on various MapReduce jobs.

However, without the equipment of a lightweight task migration strategy, many solutions, including straggler mitigation [106, 25, 23], load balancer [55, 36] that leverage task duplication are unable to achieve the optimal performance due to severe performance penalty imposed by repeating the same workload. Therefore, in future, we will continue to pursue the design of lightweight task migration mechanism so as to fully exploit the performance of various optimization techniques.

Enhancing MapReduce Schedulers for HPC Platforms: As elucidated in Chapter 5, when MapReduce systems are deployed on HPC platforms, MapReduce programs exhibit different performance behavior when retrieving input from or storing intermediate data into different storage architectures due to their distinct computation characteristics. With limited memory space available on nodes, we deem it is interesting to further study the scheduling policies that can efficiently schedule data swapping between compute nodes and storage systems. Although caching techniques have been investigated substantially, how to cache MapReduce data in the HPC platforms is not well studied. Therefore, based on the findings in Chapter 5, we will continue to enhance the schedulers of MapReduce frameworks for HPC platforms.

Bibliography

[1] Accelerating and Simplifying Apache Hadoop with Panasas ActiveStor. https://www.panasas.com/sites/default/files/uploads/docs/hadoop_wp_lr_1096.pdf.

[2] Apache Giraph. http://giraph.apache.org/.

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

[4] Bcache. http://bcache.evilpiepirate.org/.

[5] Berkeley Data Analytics Stack. https://amplab.cs.berkeley.edu/software/.

[6] Crossbow: Genotyping from short reads using cloud computing. http://bowtie-bio.source forge.net/crossbow/index.shtml.

[7] Efficient MapReduce for Big Data Analytics. http://pasl.eng.auburn.edu/doku/doku.php?id=hadoopa.

[8] Gridmix2. http://hadoop.apache.org/mapreduce/docs/ current/gridmix.html.

[9] Hadoop Capacity Scheduler. http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html.

[10] Hadoop Fair Scheduler. http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html.

[11] Hyperion Project. https://hyperionproject.llnl.gov.

[12] MapReduce-MPI Library. mapreduce.sandia.gov.

[13] Next Generation Hadoop MapReduce. http://hadoop.apache.org/docs/current.

[14] Open Fabrics Alliance. http://www.openfabrics.org.

[15] Sort Benchmark. http://sortbenchmark.org/.

[16] Test-TCP. http://www.pcausa.com/Utilities/pcattcp.htm.

[17] The 2011 Digital Universe Study: Extracting Value from Chaos. http://www.emc .com/collateral/demos/microsites/emc-digital-universe-2011/index.htm.

[18] The Public Netperf Homepage. http://www. netperf.org/ netperf/NetperfPage.html.

[19] YARN, the Hadoop Operating System. hortonworks.com/labs/yarn/.

[20] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[21] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 61–74, New York, NY, USA, 2012. ACM.

[22] Ganesh Ananthanarayanan, Christopher Douglas, Raghu Ramakrishnan, Sriram Rao, and Ion Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 24:1–24:7, New York, NY, USA, 2012. ACM.

[23] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: attack of the clones. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[24] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[25] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[26] Rajagopal Ananthanarayanan, Karan Gupta, Prashant Pandey, Himabindu Pucha, Prasenjit Sarkar, Mansi Shah, and Renu Tewari. Cloud analytics: do we really need to reinvent the storage stack? In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[27] Maltzahn Carlos, Molina-Estolano Esteban, Khurana Amandeep, J. Nelson Alex, Scott A. Brandt, and Weil Sage. Ceph as a scalable alternative to the hadoop distributed file system. ;login' 10. USENIX Association, 2010.

[28] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[29] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, August 2012.

[30] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[31] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[32] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[34] Constantinos Evangelinos and Chris N. Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2. In *In The 1st Workshop on Cloud Computing and its Applications (CCA)*, 2008.

[35] Zacharia Fadika, Elif Dede, Madhusudhan Govindaraju, and Lavanya Ramakrishnan. Adapting mapreduce for hpc environments. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 263–264, New York, NY, USA, 2011. ACM.

[36] Rohan Gandhi, Di Xie, and Y. Charlie Hu. Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 61–66, Berkeley, CA, USA, 2013. USENIX Association.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[38] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association.

[39] Abhishek Gupta and Dejan Milojicic. Evaluation of hpc applications on cloud. *Open Cirrus Summit*, 0:22–26, 2011.

[40] Mor Harchol-Balter, Nikhil Bansal, Bianca Schroeder, and Mukesh Agrawal. Srpt scheduling for web servers. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP*, volume 2221 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2001.

[41] HPC Wire. RoCE: An Ethernet-InfiniBand Love Story. http://www.hpcwire.com/blogs/.

[42] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md. Wasi ur Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High-performance design of hbase with rdma over infiniband. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, IPDPS'12, pages 774–785, 2012.

[43] Sun Microsystems Inc. Using Lustre with Apache Hadoop. http://wiki.lustre.org.

[44] InfiniBand Trade Association. The InfiniBand Architecture. `http://www.infinibandta.org/specs`.

[45] InfiniBand Trade Association. Socket Direct Protocol Specification V1.0, 2002.

[46] 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 '09, pages 261–276, New York, NY, USA, 2009. ACM.

[47] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[48] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, September 2010.

[49] Jiesheng Wu and Pete Wychoff and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Kaohsiung, Taiwan, October 2003.

[50] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, CCGRID'12, pages 236–243, Washington, DC, USA, 2012. IEEE Computer Society.

[51] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached design on high performance rdma capable interconnects. In *ICPP*, pages 743–752. IEEE, 2011.

[52] Dileep Kumar Kadali, R.N.V.Jagan Mohan, and Y. Vamsidhar. Similarity based query optimization on map reduce using euler angle oriented approach. In *International Journal of Scientific and Engineering Research*, volume 3, August 2012.

[53] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, 2010.

[54] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 181–192, New York, NY, USA, 2010. ACM.

[55] 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*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[56] Palden Lama and Xiaobo Zhou. Aroma: automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[57] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 985–996, New York, NY, USA, 2011. ACM.

[58] Boduo Li, Sandeep Tata, and Yannis Sismanis. Sparkler: supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 625–636, New York, NY, USA, 2013. ACM.

[59] Xiaobing Li, Yandong Wang, Yizheng Jiao, Cong Xu, and Weikuan Yu. Coomr: Cross-task coordination for efficient data management in mapreduce programs. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 42:1–42:11, New York, NY, USA, 2013. ACM.

[60] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming (IJPP)*, 32:167–198, 2004.

[61] Zhuo Liu, Bin Wang, Teng Wang, Yuan Tian, Cong Xu, Yandong Wang, Weikuan Yu, Carlos A. Cruz, Shujia Zhou, Tom Clune, and Scott Klasky. Profiling and improving I/O performance of a large-scale climate scientific application. In *Computer Communications and Networks(ICCCN), 2013 International Conference on*. IEEE, 2013.

[62] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing mapreduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, May 2010.

[63] Aniruddha Marathe, Rachel Harris, David K. Lowenthal, Bronis R. de Supinski, Barry Rountree, Martin Schulz, and Xin Yuan. A comparative study of high-performance computing on the cloud. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC '13, pages 239–250, New York, NY, USA, 2013. ACM.

[64] Michael Isard and Mihai Budiu and Yuan Yu and Andrew Birrell and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *EuroSys*, pages 59–72. ACM, 2007.

[65] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[66] P. H. Carns and W. B. Ligon III and R. B. Ross and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[67] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 58:1–58:11, New York, NY, USA, 2011. ACM.

[68] Xinyu Que, Yandong Wang, Cong Xu, and Weikuan Yu. Hierarchical merge for scalable mapreduce. In *Proceedings of the 2012 workshop on Management of big data systems*, MBDS '12, pages 1–6, New York, NY, USA, 2012. ACM.

[69] Jorge-Arnulfo Quiane-Ruiz, Christoph Pinkel, Jorg Schad, and Jens Dittrich. Rafting mapreduce: Fast recovery on the raft. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 589–600, Washington, DC, USA, 2011. IEEE Computer Society.

[70] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24. IEEE Computer Society, 2007.

[71] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[72] R. Recio, P. Culley, D. Garcia, and J. Hilland. An rdma protocol specification (version 1.0), October 2002.

[73] S. Sur and H. Wang and J. Huang and X. Ouyang and D. K. Panda. Can High-Performance Interconnects Benefit Hadoop Distributed File System? In *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC). Held in Conjunction with MICRO*, Dec 2010.

[74] Sarah Loebman and Dylan Nunley and YongChul Kwon and Bill Howe and Magdalena Balazinska and Jeffrey P. Gardner. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *IEEE Cluster Conference*, pages 1–10, 2009.

[75] Saba Sehrish, Grant Mackey, Jun Wang, and John Bent. Mrap: A novel mapreduce-based framework to support hpc analytics applications with access patterns. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 107–118, New York, NY, USA, 2010. ACM.

[76] 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*, pages 1–8. IEEE, 2009.

[77] Kai Shen and Stan Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 67–78, Berkeley, CA, USA, 2013. USENIX Association.

[78] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.

[79] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[80] Jian Tan, Shicong Meng, Xiaoqiao Meng, and Li Zhang. Improving reducetask data locality for sequential mapreduce jobs. In *INFOCOM*, pages 1627–1635. IEEE, 2013.

[81] Jian Tan, Xiaoqiao Meng, and Li Zhang. Delay tails in mapreduce scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint nternational conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 5–16, New York, NY, USA, 2012. ACM.

[82] Jian Tan, Yandong Wang, Weikuan Yu, and Li Zhang. Achieving Fair and Fast Completion Through WorkConserving ReduceTask Preemption. under review.

[83] Jian Tan, Yandong Wang, Weikuan Yu, and Li Zhang. Characterizing Criticality and Diffusion Limit on Fairness and Efficiency for MapReduce. SIGMETRICS '14, New York, NY, USA, 2014. ACM.

[84] 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. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[85] Yuan Tian, Scott Klasky, Hasan Abbasi, Jay F. Lofstead, Ray W. Grout, Norbert Podhorszki, Qing Liu, Yandong Wang, and Weikuan Yu. Edo: Improving read performance for scientific applications through elastic data organization. In *CLUSTER*, pages 93–102. IEEE, 2011.

[86] D. N.C. Tse, R. G. Gallager, and J. N. Tsitsiklis. Statistical multiplexing of multiple time-scale markov streams. *IEEE J.Sel. A. Commun.*, 13(6):1028–1038, September 2006.

[87] 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*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

[88] Weina Wang, Kai Zhu, Lei Ying, Jian Tan, and Li Zhang. A throughput optimal algorithm for map task scheduling in mapreduce with data locality. *SIGMETRICS Performance Evaluation Review*, 40(4):33–42, 2013.

[89] Yandong Wang, Yizheng Jiao, Cong Xu, Xiaobing Li, Teng Wang, Xinyu Que, Cristian Cira, Bin Wang, Zhuo Liu, Bliss Bailey, and Weikuan Yu. Assessing the performance impact of high-speed interconnects on mapreduce. In Tilmann Rabl, Meikel Poess, Chaitanya K. Baru, and Hans-Arno Jacobsen, editors, *WBDB*, volume 8163 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2012.

[90] 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*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[91] Yandong Wang, Jian Tan, Weikuan Yu, Xiaoqiao Meng, and Li Zhang. Preemptive reducetask scheduling for fair and fast job completion. In *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC'13, June 2013.

[92] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey balmin. Flex: a slot allocation scheduling optimizer for mapreduce workloads. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[93] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[94] Cong Xu. Tcp/ip implementation of hadoop acceleration. In *Auburn Theses and Dissertations*, 2012.

[95] Cong Xu, Manjunath Gorentla Venkata, Richard L. Graham, Yandong Wang, Zhuo Liu, and Weikuan Yu. Sloavx: Scalable logarithmic alltoallv algorithm for hierarchical multicore systems. In *CCGRID*, pages 369–376. IEEE Computer Society, 2013.

[96] Yandong Wang and Cong Xu and Xiaobing Li and Weikuan Yu. JVM-Bypass shuffling for hadoop acceleration. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[97] Yandong Wang and Robin Goldstone and Weikuan Yu and Teng Wang. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In *28th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2014)*, Phoenix, USA, May 2014.

[98] Weikuan Yu, Shuang Liang, and Dhabaleswar K. Panda. High performance support of parallel virtual file system (pvfs2) over quadrics. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 323–331, New York, NY, USA, 2005. ACM.

[99] Weikuan Yu, Yandong Wang, and Xinyu Que. Design and evaluation of network-levitated merge for hadoop acceleration. In *IEEE Transactions on Parallel and Distributed Systems*, pages 602–611, 2013.

[100] Weikuan Yu, Yandong Wang, Xinyu Que, and Cong Xu. Virtual shuffling for efficient data movement in mapreduce. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013.

[101] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[102] 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*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[103] 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'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[104] 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*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[105] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[106] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[107] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 53–62, New York, NY, USA, 2012. ACM.