Scalable Collective Communication and Data Transfer for High-Performance Computation and Data Analytics

by

Cong Xu

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

Auburn, Alabama
May 10, 2015

Keywords: MPI, High Performance Computing,
Cloud Computing, Big Data, MapReduce, Analytics Shipping

Abstract

Large-scale computer clusters are leveraged to solve various crucial problems in both High Performance Computing and Cloud Computing fields. Message Passing Interface (MPI) and MapReduce are two prevalent tools to tap the power of parallel data processing and computing resources in HPC and commercial machines. Scientific applications use collective communication operations in MPI for global synchronization and data exchanges. Meanwhile, MapReduce is a popular programming model that provides a simple and scalable parallel data processing framework for large-scale off-the-shelf clusters. Hadoop is an open source implementation of MapReduce and YARN is the next-generation of Hadoop's compute platform. To achieve efficient data processing over large-scale clusters, it is crucial to improve MPI collective communication and optimize the MapReduce framework.

However, the existing MPI libraries and MapReduce framework face critical issues in terms of collective communication and data movement. Specifically, the MPI AlltoallV operation relies on a linear algorithm for exchanging small messages, it cannot obtain the advantage of shared memory on hierarchical multicore system. Meanwhile, Hadoop employs Java-based network transport stack on top of the Java Virtual Machine (JVM) for its MapTask to ReduceTask all to all data shuffling and merging purposes. Detailed examination reveals that JVM imposes a significant amount of overhead to data processing. In addition, scientific datasets are stored on HPC backend storage servers, these datasets can be analyzed by Yarn MapReduce programs on compute nodes. However, the storage servers and computation powers are separated in the HPC environment, the datasets are too costly to transfer due to their sheer size.

This dissertation has addressed above issues in the existing MPI libraries and MapReduce framework. Accordingly, the MPI collective communication and Hadoop data movement have been optimized. For MPI AlltoallV collective operation, we design and implement a new *S*calable

*LO*garithmic *A*lltoall*V* algorithm, named *SLOAV*, for MPI AlltoallV collective operations. SLOAV aims to achieve global exchange of small messages of different sizes in logarithmic manner. Furthermore, we design a hierarchical AlltoallV algorithm based on SLOAV by taking advantage of shared memory in multicore systems, which is referred to as *SLOAVx*. Also this dissertation has optimized Hadoop with *J*VM *B*ypass *S*huffling *(JBS)* plugin library for fast data movement, overcoming the existing limitations, and removing the overhead and limitations imposed by JVM. In the third study, we exploit the analytics shipping model for fast analysis of large-scale scientific datasets on HPC backend storage servers. Through an efficient integration of MapReduce and the popular Lustre storage system, we have developed a *V*irtualized *A*nalytics *S*hipping *(VAS)* framework that can ship MapReduce programs to Lustre storage servers.

Acknowledgments

My Ph.D research study would not have made progress without the support of a lot of people. Firstly, I'd like to express my special gratitude to my advisor, Dr. Weikuan Yu, for his kind guidance, encouragement and patience in achieving the goals of my Ph.D research. I have to say, Dr. Yu has made available his support in a number of ways. He not only gives instructions on my research projects but also continues to encourage my spirit, especially when I got involved in some difficulties. I really want to express my sincere appreciation for his help. It is an honor for me to be his student.

In addition, I would like to gratefully and sincerely thank Dr. Manjunath Gorentla Venkata and Dr. Richard L. Graham for their guidance, understanding, patience and most importantly, their friendship during my summer internship at Oak Ridge National Laboratory. I also appreciate Ms. Robin Goldstone from Lawrence Livermore National Laboratory and Bryon Neitzel from Intel, who provided great help on the Analytics Shipping project.

Furthermore I also want to thank all of the professors and staff members in the Computer Science and Software Engineering Department who have kindly helped me in these years. Specifically, I'd like to express my gratitude to our department Chair Dr. Kai Chang for his invaluable endorsement and guidance. I would also like to acknowledge my advisory committee members, Dr. Dean Hendrix, Dr. Saad Biaz and my outside reader Dr. Wenxian Shen for their time and patience on giving suggestions on my thesis dissertation.

I am indebted to my colleagues in the Parallel Architecture and System Laboratory (PASL) at Auburn University: Yandong Wang, Zhuo Liu, Xinyu Que, Bin Wang, Hui Chen, Jianhui Yue, Yuan Tian, Teng Wang, Xiaobin Li, Huansong Fu, Fang Zhou, Xinning Wang, Kevin Vasko and Michael Pritchard, who have helped me a lot on my research work. They always make me feel that we are one family. I never feel lonely.

Finally, I would like to express my sincere thanks to my father Yingjie Xu my mother Yan Liu. I always have them behind me, and there’s a lot of love, a lot of support and it’s just great.

Table of Contents

Chapter 1

Introduction

In High-Performance Computing (HPC) environment, supercomputers are leveraged by large-scale scientific applications to solve all kinds of large-scale scientific problems, such as Climate Modeling, Flight Vehicle Designing and so on. Scientific applications execute their work on the compute nodes using multiple processes in a parallel manner. These processes communicate with each other by using Message Passing Interface (MPI) [8]. To speed up scientific application computation, it is critical to improve MPI collective communication.

The successful emergence of Cloud Computing has radically changed the IT industry trends by providing computational resources on demand via a computer network. In cloud computing, MapReduce [22] is considered one of the most popular programming models for processing and analyzing large data sets. Users are able to first create a function handling a Map based on key/value pair collection of data; then create a Reduce function to combine all of the intermediate key values with the same value of the intermediate value. The framework of the MapReduce program can be executed in parallel on a large number of computers, and this system focuses on how to split the input data, schedule the execution of programs on a large number of machines, deal with machine failures, and manage the communication between computers. In order to improve the performance of processing large scale data sets, it is imperative to optimize MapReduce.

Given the fact that integrating traditional HPC Grid services with on-demand Cloud services is able to provide more flexibility in HPC and Cloud markets, achieving efficient integration is important. HPC on supercomputers has been a widely leveraged paradigm to tap the power of parallel data processing and computing resources for Cloud Computing. Because of the sheer size of scientific data, it is very costly to move that data back and forth on HPC systems for the purpose of analytics or visualization. We take on an effort to exploit the analytics shipping model for fast analysis of large-scale persistent scientific datasets. In this dissertation, we use MapReduce and Lustre as a representative analytics model and HPC backend storage system.

This section is going to provide some brief introductions in the areas of HPC and Cloud Computing. For HPC systems, typical HPC Supercomputer Architecture, MPI Collective Operation, and Lustre filesystems [6] will be presented. In Cloud Computing Environment, we are going to explain MapReduce programming model and its implementations Hadoop [2] and Yarn [1].

1.1 Introduction to MPI Collective Operation in HPC Environment

MPI is short for Message Passing Interface, it is a library standard defined by a committee of vendors, implementers and parallel programmers. MPI is used to create parallel programs based on message passing. Normally the same program is running on several different cores, different processes execute their work and communicate with each other using message passing.

MPI_Alltoall and MPI_Alltoallv are two important collective operations, in which every process exchanges messages with all of processes participated in current MPI communication. MPI_Alltoall is adopted to exchange same data type and same element size while MPI_Alltoallv copes with same data type but different element size. This dissertation mainly focuses on optimizing MPI_Alltoallv small message exchanging.

1.2 Overview of Hadoop MapReduce Framework in Cloud Computing

MapReduce is a new programming model that can hide the complexity of data distribution, parallel programming, fault tolerance and load balance from the user, it provides a simple and scalable parallel data processing framework for large-scale off-the-shelf clusters. Hadoop is an open source implementation of MapReduce framework, it exposes two simple interfaces: *map* and *reduce*, to application users but hides processing complexities, such as data distribution, task parallelization, fault tolerance, etc. Its runtime system consists of four major components: JobTracker, TaskTracker, MapTask, and ReduceTask. These components are shown in Figure 1.1. JobTracker assigns one TaskTracker per slave node and orchestrates TaskTrackers to launch MapTasks and

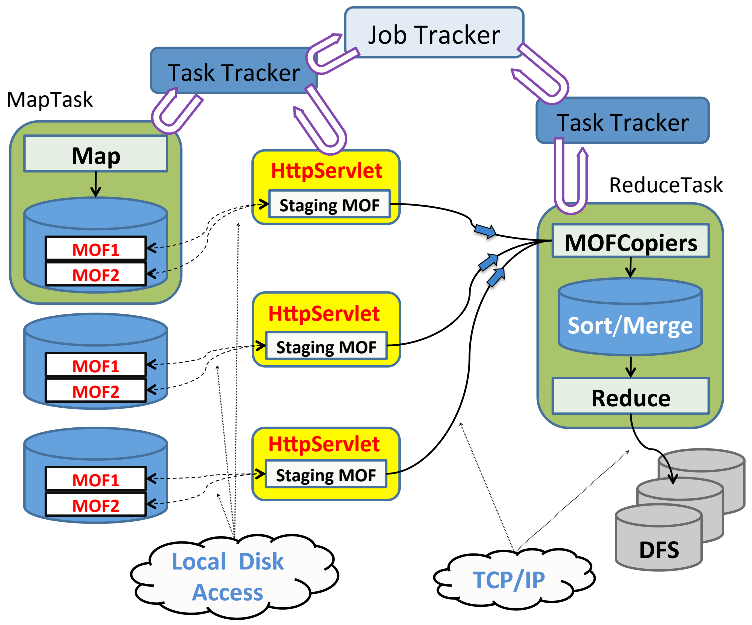


Figure 1.1: Details of Intermediate Data Shuffling

ReduceTasks for job execution. A MapTask reads an input split from Hadoop Distributed File System (HDFS) [53], runs the map function, and stores intermediate data as a **M**ap **O**utput **F**ile (MOF) to local disks. A MOF is divided into multiple *segments*, each of which is for a specific ReduceTask. Each ReduceTask fetches the segments from all the MOFs, sorts/merges them, and then reduces the merged results. The output generated by a ReduceTask is stored back to the HDFS as a part of the final result.

Hadoop has been highly optimized to reduce the amount of network traffic when reading input data for MapTasks and writing output from ReduceTasks. For instance, delay scheduling [71] helps improving the data locality and reducing data movement in the network. According to [71], up to 98% of 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 HDFS.

However, Hadoop intermediate data shuffling still causes a large volume of network traffic. 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 [48] 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, Hadoop performance degrades non-linearly with the increase of intermediate data size. As pointed out by [21], 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 Hadoop.

Hadoop currently relies on a stack of transport protocols in the Java Virtual Machine (JVM), including Java HTTP and network libraries. However, JVM introduces significant overhead in managing Java objects. For example, for every 8-byte *double* object, it requires another 16 bytes for data representation, an overhead of 67% [43]. Such inflated memory consumption quickly shrinks the available memory to Hadoop, and prolongs Java garbage collection for reclaiming memory. This JVM issue has also been documented by [14, 34, 51].

Contemporary high speed networks, such as InfiniBand [31], provide Remote Direct Memory Access (RDMA) [49] that is capable of up to 56Gbps bandwidth, sub-microsecond latency and low CPU utilization. RDMA is also available through the RoCE (RDMA over Converged Ethernet) protocol [28] on 10 Gigabit Ethernet (10GigE). The performance advantage of RDMA advocates it as a compelling solution to speed up global intermediate data shuffling in Hadoop. Unfortunately, the existing Hadoop is designed to completely rely on the legacy TCP/IP protocol to transfer intermediate data and is incapable of utilizing RDMA. Such limitation prevents Hadoop from relishing high bandwidth and low latency from InfiniBand and 10GigE networks which have been demonstrated as effective data movement technologies in many networking environments due their low-latency, high bandwidth, and low CPU utilization.

With so many problems in current Hadoop architecture, it is a critical issue to examine the entire Hadoop framework, particularly, its intermediate data shuffling, so that Hadoop MapReduce can make the best use of the resources provided by the clusters that consists of state-of-the-art commodity machines. To address above critical issues in Hadoop MapReduce framework, **J**VM-**B**ypass **S**huffling (JBS) is introduced for Hadoop to avoid the overhead imposed by JVM during data shuffling and enable a portable framework for fast data movement on both RDMA and TCP/IP protocols.

1.3 An Overview of YARN MapReduce

YARN is the next-generation of Hadoop's compute platform [59, 1]. It can support various programming models such as MapReduce and MPI. There are three main components in the YARN MapReduce framework, including the ResourceManager (RM), NodeManager (NM), and the ApplicationMaster (AM). The RM is responsible for allocating resources, which is abstracted as *containers*, to applications. A NM monitors and reports its container's resource usage (cpu, memory, disk, network) to RM. The per-application AM is responsible for the job's whole life cycle management including resource negotiating, task deployment, status monitoring and reporting, till the application exits.

A MapReduce job starts an AM that negotiates with the RM for containers. Once the containers are allocated, the AM will communicate with the corresponding NMs to launch map tasks (MapTasks). Each MapTask reads an input split from the HDFS[53] and generates intermediate data in the form of Map Output Files (MOFs). Each MOF contains as many partitions as the number of reduce tasks (ReduceTasks). ReduceTasks are launched after the generation of some MOFs. Each fetches its partition from every MOF. Meanwhile, the background merging threads will merge and spill fetched intermediate data to local disks. In the reduce phase, the merged intermediate data will be processed by the reduce function for final results, which will be written back to HDFS.

For minimizing data movement, MapReduce adopts a data-centric paradigm to co-locate compute and storage resources on the same node to facilitate locality-oriented task scheduling. In YARN's case, a job's MapTasks will be scheduled to the containers where the input data splits are located.

1.4 Overview of HPC and Lustre Backend Storage System

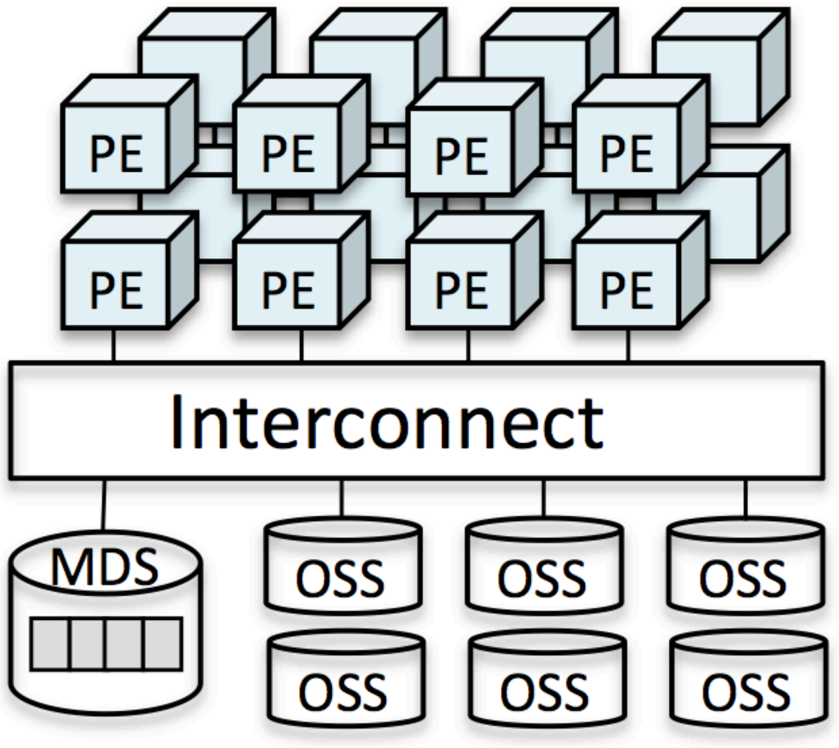


Figure 1.2: Diagram of a Typical HPC System Architecture

Fig. 1.2 shows a diagram of typical HPC systems. Such systems are constructed using a compute-centric paradigm where compute nodes and storage servers belong to separated groups. 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. With such compute-centric paradigm, tasks on compute nodes of HPC systems are, in general, equally distant from the backend storage system.

Lustre is a popular backend storage system used on many HPC systems. A typical Lustre file system consists of metadata server (MDS), metadata target (MDT), object storage servers (OSS), object storage target (OST) and Lustre clients. The MDS makes metadata stored in one or more MDTs available to Lustre clients. Each MDS manages the names and directories in the Lustre file system and provides network request handling for local MDTs. The OSS provides file I/O service, and network request handling for one or more local OSTs. A typical configuration is an MDT on a dedicated node, two or more OSTs on each OSS node.

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 also on the amount of lock contention among them.

Chapter 2

Related Work

This chapter is going to present previous work related to MPI AlltoallV collective operation, Hadoop data shuffling, and research on analytics shipping on HPC Backend Storage Servers. Furthermore, the distinction between existing work and our efforts are also mentioned here.

2.1 Studies on Improving Alltoallv Communication in HPC Envirionment

Collective communication has been extensively researched in High-Performance Computing (HPC), however, very little research has been conducted for AlltoallV collective communications. In addition, none of the early work has studied AlltoallV communication with logarithmic complexity. Jackson and Booth [33] have proposed *Planned AlltoallV* to optimize AlltoallV in the clustered architecture. Their optimization collects data into one single message from all processes on the same node before conducting inter-node communication, so that the number of messages sent between different nodes can be dramatically reduced. Goglin *et al.* [16] proposed a kernel-assisted memory copy module (KNEM). It can bring benefits to collective intra-node AlltoallV communication. Later on, Ma *et al.* [41] optimized the KNEM by making use of memory architecture on NUMA architecture. However, the complexity of above algorithms and optimizations still require linear complexity to accomplish AlltoallV communication.

Brightwell and Underwood [17] carried out a deep analysis of the advantages of leveraging offload in MPI collectives to overlap the communication and computation and demonstrated the performance improvements for the NAS Parallel Benchmark [7]. However, the performance of MPI_Alltoallv was not improved. Faraj and Yuan [24] optimized MPI programs by leveraging *compiled communication*, taking advantage of the compiler's knowledge of network architecture and application communication requirements. The effectiveness has also been shown on NAS

Parallel benchmark [7]. However, it was only effective for static communication with fixed patterns at compilation time. This work was unable to optimize dynamic communications, such as AlltoallV operation, since the compiler is unable to do array analysis. Plummer and Refson [46] optimized the MPI AlltoallV for materials science code CASTEP [19], their approach is to breakdown AlltoallV into multiple groups of processors and only require the group leader to participate in AlltoallV communication. However, the improvement is limited due to linear complexity and the organization of processes on each node can cause performance bottleneck.

Recursive Doubling (RDB) [68] and Bruck [18] algorithms are two logarithmic algorithms used in Alltoall communication to exchange small messages. When the process number is power of two, Bruck algorithm sends fewer amounts of data in comparison to RDB, and it works much better than RDB in realistic cases. However both of them cannot support messages of variant sizes.

2.2 Existing Work on Optimizing Hadoop Data Shuffling

Leveraging high performance interconnects to move data in the Hadoop ecosystem has attracted numerous research interests from many organizations. Huang *et al.* [29] designed an RDMA-based HBase over InfiniBand. In addition, they also mentioned the disadvantages of using Java Socket Interfaces. Jose *et al.*[36, 35] implemented a scalable memcache through taking advantage of performance benefits provided by high-speed interconnects. Sur *et al.* [54] studied the potential benefit of running HDFS over InfiniBand. Furthermore, Islam *et al.* [32] enhances the HDFS using RDMA over InfiniBand via JNI interfaces. However, 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.

Solving the intermediate data shuffling bottleneck is another interesting research topic about Hadoop. Camdoop [21] 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. MapReduce online [20] attempts 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. Different from MapReduce online, JBS completely re-designs both server and client sides and eliminates the JVM overhead associated with the data shuffling. Seo *et al.* [50] improved the performance of MapReduce by reducing redundant I/O in the software architecture. But it did not study the I/O issues caused by the data shuffling between MapTasks and ReduceTasks. [30] replaces HDFS with high-performance Lustre file system, and stores intermediate data in Lustre. However there is no efficient performance improvement reported.

Leveraging RDMA from high speed networks for high-performance data movement has been very popular in various programming models and storage paradigms. [26] studied the pros and cons of using RDMA capabilities. Liu *et al.* [38] designed RDMA-based MPI over InfiniBand. Yu *et al.* [67] implemented a scalable connection management strategy for high-performance interconnects. Our work is based on these mature studies of RDMA technology.

2.3 Research on Analytics Shipping on HPC Backend Storage Servers

The *active storage* implementation in [45] was also designed to exploit the idle computation power in the storage nodes for running some general programs. Differently, our VAS framework leveraged the MapReduce programming model for parallel and efficient data analysis.

In order to avoid enormous data migration back and forth in the supercomputer, there have been many studies providing *in-situ* data analytics along with scientific simulations [63, 72]. For example, Tiwari et al. [58] implemented an active flash prototype to conduct analysis on the solid-state drives before simulation results are stored back to disks. In addition, some other analysis techniques have been developed for better coordination with the large-scale scientific simulations. Among them, Bennett et al. [15] exploited the DataSpaces and ADIOS frameworks for supporting efficient data movement between *in-situ* and *in-transit* computations. In general, *in-situ* analysis and visualization have become one important trend of data interpretation in large-scale scientific simulations. However, for the data already stored in backend storage system, it remains a big challenge to conduct such data analysis efficiently.

Leveraging advanced data analytics frameworks, such as MapReduce, is a promising solution. There have been several attempts of integrating MapReduce based data analytics model into compute-centric High Performance Computing environment to work with distributed file systems. For example, Ananthanarayanan et al. [13] compared the performance of HDFS with a commercial cluster filesystem (IBM's GPFS) for a variety of MapReduce workloads. Maltzahn et al. [42] showed the feasibility of executing Hadoop with the Ceph file system. Tantisiriroj et al. [55, 56] integrated PVFS with Hadoop and compared its performance with Hadoop on HDFS. In [40], performance studies were conducted to compare Hadoop with HDFS and Hadoop with Lustre. The results showed that Hadoop with Lustre performed significantly worse than Hadoop with HDFS mainly due to the inefficient intermediate data access and processing on Lustre. In most cases, integrating Hadoop with parallel file systems has so far demonstrated unsatisfactory performance. That is because porting Hadoop to HPC systems may suffer from issues such as resource contention and performance interference, etc. Our virtualized analytics shipping framework on Lustre is able to address such issues through a set of new techniques and demonstrate more efficient analysis than Hadoop on HDFS.

Chapter 3

Problem Statement

This chapter details the issues and challenges addressed in this dissertation. Firstly, I have illustrated the limitation of the existing AlltoallV algorithm. Following that, I also characterize and analyze two critical issues that prevent Hadoop runtime from achieving high throughput. Finally, I have undertaken an effort to systematically examine a series of challenges for effective analytics shipping.

3.1 Limitation of Existing AlltoallV Algorithm

Existing MPI AlltoallV implementation has linear complexity, i.e., each process has to send messages to all other processes in the job. The number of required messages to accomplish AlltoallV increases linearly with the number of processes involved in the communication. Such linear complexity can result in suboptimal scalability of MPI applications when they are deployed on millions of cores.

Bruck algorithm emerged as a logarithmic algorithm to conduct Alltoall for small messages. In this algorithm, three imperative steps are carried out sequentially. Assume that the total number of processes that participate in the Alltoall communication is N and the rank of current process is n. In Step 1, the *Local Rotation Step*, message elements are rotated inside each process to prepare data for the next Step 2. Then in Step 2, inter-process communication is performed for $\lceil log_2 N \rceil$ rounds. At each round, data elements for the same destination in one process are merged before being sent out. In round s (starting from 1), process n sends data to process $(n+2^{s-1})\, mod\, N$ and receives data from process $(n-2^{s-1}+N)\, mod\, N$. After $\lceil log_2 N \rceil$ rounds of communications, all the data elements can arrive at their final destination process. In Step 3, another intra-process rotation is proceeded to relocate the data elements to correct positions.

However, Bruck algorithm cannot deal with data elements of different sizes because the length of data input buffer used in this algorithm is fixed. We term the segment for holding an initial element in the data input buffer as an *element segment*. Such algorithm cannot be directly applied to AlltoallV because during the inter-process communication, the received intermediate data element can be much larger than the capacity of the element segment, thus making the communication inapplicable.

Multicore systems with shared memory are becoming ubiquitous in supercomputing infrastructure. Efficient usage of shared memory can dramatically mitigate the overhead of network message latency during the collective communication. However, existing AlltoallV implementation has not taken advantage of shared memorys benefits on Multicore systems.

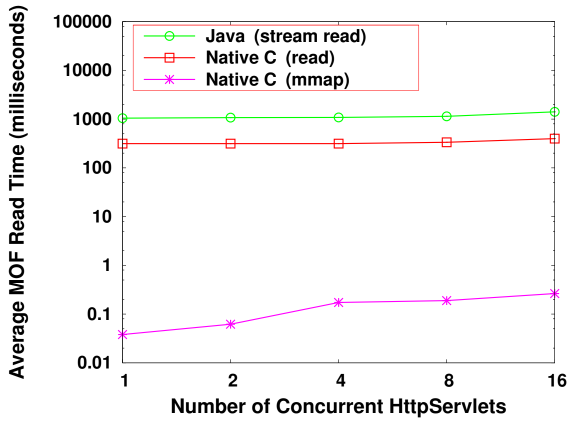
3.2 Issues in Original Hadoop Data Shuffling

Original Hadoop data shuffling is implemented by Java programming language. However, such implementation introduces two important issues: 1. Java Virtual Machine imposes significant overhead on Hadoop Data Shuffling. 2. Java is incapable of leveraging High-Performance Network Protocols. Details will be described in the following section.

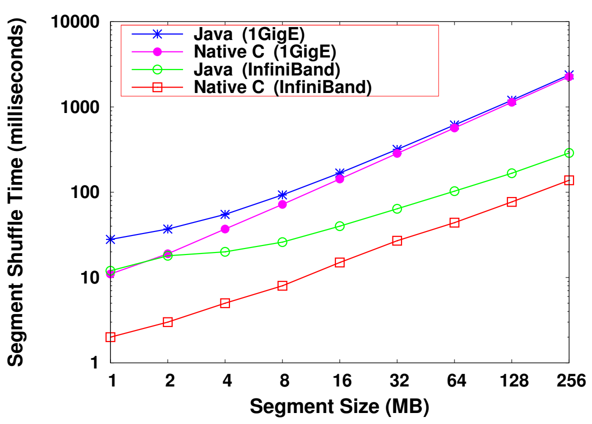
3.2.1 Overhead Imposed by Java Virtual Machine

Current Hadoop completely relies on Java Vritual Machine to conduct intermediate data movement, as depicted in Figure 1.1 (shown in section 1.2). An HttpServer is embedded inside each TaskTracker, and this server spawns multiple HttpServlets to answer incoming fetch requests for segment data. On the other side of the data shuffling, within each ReduceTask, multiple MOF-Copiers are running concurrently to gather all segments.

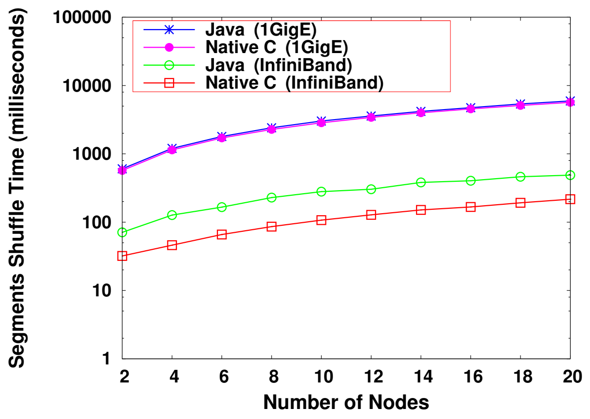
Inside HttpServlets and MOFCopiers, Hadoop employs Java streams to simultaneously access and move data. However, such kind of Java I/O can perform 40%$\sim$60% worse than that written in native C, as pointed by [51, 23]. In order to shed a light on the overhead imposed by Java Virtual Machine (JVM) on Hadoop intermediate data shuffling, we have examined data movement



(a) Disk I/O



(b) *One* HttpServlet to *One* MOFCopier



(c) *N* Nodes to *One* ReduceTask

Figure 3.1: Simulation Results of Intermediate Data Shuffling

between HttpServlets and MOFCopiers using both Java and native C languages. Figure 3.1(a) shows the performance of disk I/O using Java and native C. Currently Hadoop HttpServlets use traditional Java FileInputStreams to retrieve content from Map output. As shown in Figure 3.1(a), on average, using Java-based HttpServlets to read MOFs can be 3.1 $\times$ worse than using native C language.

Figure 3.1(b) shows the time to shuffle data between one HttpServlet and one MOFCopier. On 1 Gigabit Ethernet (1GigE), the constraining effect of JVM is hidden because of the limited network bandwidth. On InfiniBand, Java-based data shuffling leads to a performance degradation of as much as 3.4$\times$, compared to native C. Furthermore, Figure 3.1(c) shows that on the InfiniBand cluster, when one ReduceTask (with multiple MOFCopiers) is fetching segments simultaneously from multiple nodes, JVM imposes above 2.5× overhead. Again, this overhead is hidden when the network bandwidth is bottlenecked on 1GigE.

Above results adequately demonstrate that JVM can impose significant overhead on the Hadoop intermediate data shuffling, thus preventing the Hadoop from fully exploiting the performance provided by the underlying systems, when both sender and receiver machines can provide enough resources.

3.2.2 Incapability of Leveraging High-Performance Network Protocols

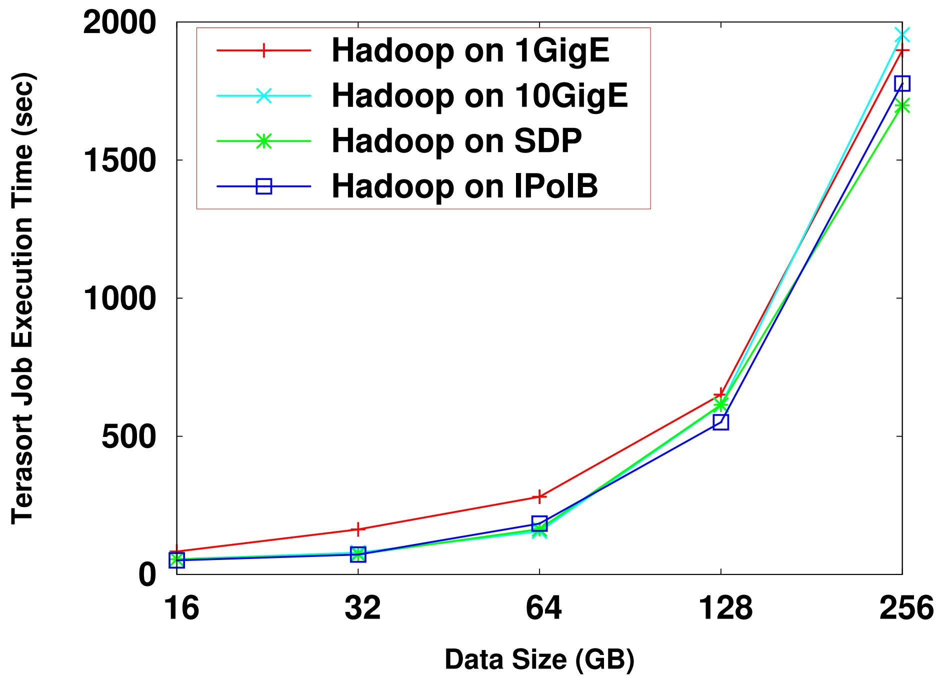


Figure 3.2: Running Current Hadoop on Different Networks

Besides the TCP/IP protocol, current Hadoop does not support other transport protocols such as RDMA on InfiniBand [31] and RoCE (RDMA over Converged Ethernet) on 10Gigabit Ethernet (10GigE) that have matured in the HPC (High Performance Computing) community. Simply replacing the network hardware with the latest interconnect technologies such as InfiniBand and 10GigE, and continuing to run Hadoop on TCP/IP will not enable Hadoop to leverage the strengths of RDMA and cannot effectively acceleration the execution of Hadoop jobs. As shown in Figure 3.2, compared to Hadoop on 1GigE, running Hadoop over InfiniBand through IPoIB and SDP protocols and 10GigE can only moderately reduce job execution. It is also worth noting that despite the high price differential between RDMA capable interconnects and traditional commodity Gigabit Ethernets, such differential has shrunk significantly over the past few years. Many RDMA capable interconnects, including InfiniBand and 10GigE, are becoming popular commodity products. Thus, the lack of portability on multiple interconnects will prevent Hadoop from keeping 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.

Figure 3.2 shows the performance of running original Hadoop under different networks with different protocols. As shown in the Figure, compared to running Hadoop over 1GigE, simply running Hadoop over high-performance interconnects and continuing using TCP/IP protocol can only moderately improve the performance when data size is small ($\leq$ 64GB). Although SDP provides Java applications with the opportunity to relish the RDMA advantages provided by InfiniBand, running Hadoop over SDP achieves no noticeable performance improvement when compared to running over IPoIB. These results confirm that existing Hadoop is incapable of fullying leverage the performance advantages provisioned by state-of-the-art network technologies.

3.3 Challenges for Shipping Analytics to Backend Storage Servers

In this section, we discuss the challenges associated with shipping analytics to backend storage servers on HPC systems.

3.3.1 Isolating Analytics from I/O Services

Backend storage servers on HPC systems are typically dedicated to I/O services. Shipping analytics to these servers will inevitably disrupt this dedication. Concerns can arise from a variety of aspects, including resource contention, performance interference and integrity and availability of data hosted by the storage servers. There is a serious need to isolate analytics from I/O services when the backend storage servers are used to host both analytics programs and I/O services. In this paper, we consider the users and their analytics programs to the HPC systems are trustworthy, which is a reasonable assumption considering the rigidity and secure practices in allocating user accounts and computer time allocations on HPC systems. Thus we focus on the issues of resource contention and performance interference.

There are many approaches such as chroot [3], virtualization [10, 11, 5] to isolating analytics from I/O services. The most appropriate approach should allow all the flexibility of scheduling and tuning analytics programs on the backend storage servers while, at the same time, providing rich mechanisms to alleviate resource contention and mitigate performance interference. For its rich functionalities and versatile execution, we consider the use of virtual machines to be the most suitable for analytics shipping. While virtualization provides handy mechanisms for isolating analytics and segregating allocated resources, it is critical to have a thorough examination on its performance implication to both I/O services and analytics and accordingly mitigate the performance overhead where applicable.

3.3.2 Distributing Analytics Data and Exploiting Task Locality

As reviewed in Section 1.4, HPC systems and their storage servers are deployed using a compute-centric paradigm, which is distinctly different from the data-centric paradigm that is adopted by the MapReduce-style analytics model. Therefore, we need to address the mismatches of data distribution and task scheduling between MapReduce-based programs and backend storage servers. First, in terms of data distribution, the datasets of analytics programs need to be distributed in a manner that can closely match the pattern used by MapReduce. In the case of YARN, we need to emulate a distribution pattern of HDFS such that analytics datasets are split into blocks and distributed to all Lustre storage servers. Second, in terms of task scheduling, we need to extract the knowledge of data distribution and bestow this information to YARN for its scheduler to launch analytics tasks with the best data locality.

3.3.3 Intermediate Data Placement and Shuffling

Another key difference between the execution of MapReduce programs and the configuration of HPC systems is the management of intermediate data. MapReduce systems such as YARN are usually configured with local disks attached to the compute nodes. As reviewed in Section 1.4, this allows MapReduce programs to store their intermediate data temporarily on local disks. But an HPC storage system such as Lustre is usually a shared file system. This means that MapReduce programs, when shipped to Lustre servers, have Lustre as a global storage space to store any intermediate data.

Intuitively, placing intermediate data as MOFs on Lustre seems to bring a lot of convenience because all MOFs are globally available to any MapReduce task. ReduceTasks no longer need to have a shuffling stage to fetch data segments from the MOFs. However, this can have several ramifications. First, the generation of many temporary MOFs can present a huge burst of metadata operations to the often bottlenecked Lustre MDS. Second, the striping pattern of a MOF, i.e., the placement of its stripes, needs to be decided carefully. Should all stripes be placed on the Lustre OST attached to the same node with the MapTask, on another OST closer to some reduce task, or across all OSTs? Finally, when many ReduceTasks are reading from the same MOFs on Lustre, they can cause a burst of lock operations. The resulting lock contention may significantly degrade the speed of data shuffling, causing an overall performance loss.

Taken together, we need to address all challenges for an effective analytics shipping framework.

3.4 Research Contributions

To address abovementioned issues, this dissertation has proposed a couple of techniques to improve the efficiency of communication and data analytics on HPC and Cloud systems. In summary, the efforts described in this dissertation have following contributions:

- Speeding up MPI AlltoallV collective communication;

- Accelerating data shuffling in Hadoop framework; and
- Achieving efficient data analytics in HPC systems.

During my Ph.D study I spent all my efforts on solving these problems and have delivered the following publications:

1. Cong Xu, Robin Goldsone, Zhuo Liu, Hui Chen, Bryon Neitzel, Weikuan Yu. Exploiting Analytics Shipping with Virtualized MapReduce on HPC Backend Storage Servers, TPDS 2015 [65]

2. Cong Xu, Manjunath G. Venkata, Richard L. Graham, Yandong Wang, Zhuo Liu, Weikuan Yu. SLOAVx: Scalable LOgarithmic AlltoallV Algorithm for Hierarchical Multicore Systems, CCGrid 2013 [66]

3. Yandong Wang, Cong Xu, Xiaobing Li, Weikuan Yu. JVM-Bypass for Efficient Hadoop Shuffling, IPDPS 2013 [62]

4. Xiaobing Li, Yandong Wang, Yizheng Jiao, Cong Xu, Weikuan Yu. CooMR: Cross-Task Coordination for Efficient Data Management in MapReduce in Programs, SC 2013 [37]

5. Yuan Tian, Cong Xu, Weikuan Yu, Scott Klasky, Honggao Liu. neCODEC: Nearline Data Compression for Scientific Applications, CLUSTER JOURNAL 2013 [57]

6. 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, WBDB 2013 [60]

7. Weikuan Yu, Yandong Wang, Xinyu Que, Cong Xu. Virtual Shuffling for Efficient Data Movement in MapReduce, IEEE Transactions on Computers, TC 2013 [69]

8. Zhuo Liu, Bin Wang, Teng Wang, Yuan Tian, Cong Xu, Yandong Wang, Weikuan Yu, Carlos A. Cruz, Shujia Zhou, Tom Clune, Scott Klasky. Profiling and Improving I/O Performance of a Large-Scale Climate Scientific Application, ICCCN 2013 [39]

9. Xinyu Que, Yandong Wang, Cong Xu, Weikuan Yu. Hierarchical Merge for Scalable MapReduce, MBDS 2012 [47]

10. Cong Xu. Tcp/ip implementation of hadoop acceleration, Master Thesis [64]

11. Weikuan Yu, K. John Wu, Wei-Shinn Ku, Cong Xu, Juan Gao. BMF: Bitmapped Mass Fingerprinting for Fast Protein Identification, Cluster 2011 [70]

Chapter 4
System Design

To address the issues described in chapter 3, I introduce SLOAV, SLOAVx algorithms, JVM-Bypass Shuffling (JBS) library, and VAS framework. SLOAV aims to achieve global exchange of small messages of different sizes in a logarithmic number of rounds. Furthermore, given the prevalence of multicore systems with shared memory, we design a hierarchical AlltoallV algorithm based on SLOAV by leveraging the advantages of shared memory, which is referred to as SLOAVx. JVM-Bypass Shuffling library aims to eliminate the JVM from the critical path of Hadoop intermediate data movement, thus avoiding the overhead imposed by JVM. To minimize the data movement between HPC compute nodes and backend storage servers during the analysis phase, a Virtualized Analytics Shipping (VAS) framework is proposed to ship MapReduce analytics programs to Lustre storage servers. In addition, three techniques have been implemented in this VAS framework to achieve efficient analytics shipping. The rest of this chapter elaborates on the design details of our efforts.

4.1 Scalable LOgarithmic AlltoallV Algorithm for Hierarchical Multicore Systems

4.1.1 SLOAV: A Scalable LOgarithmic AlltoallV Algorithm

Traditional MPI logarithmic collective algorithms, such as Bruck algorithm, can only work for Alltoall operation whose message sizes are uniform. In this section, we introduce our new *S*calable *LO*garithmic *A*lltoall*V* algorithm (SLOAV) for processing AlltoallV collective communication.

Our work is built on top of Cheetah [27], which is a collective communication framework embedded in Open MPI. In this framework, the MPI-level communication is controlled by a component in the Multi-Level (ML) manager, named COLL in Open MPI. Subgrouping (SBGP) component is used to extract topology information. At the communicator creation time, COLL takes advantage of SBGP to discover communication hierarchies and makes use of the subgroup information generated by SBGP to do collective communication within subgroups using the BCOL (Basic Collectives framework).

4.1.1.1 Scalable Logarithmic AlltoallV Algorithm (SLOAV)

In this section, we firstly describe the data structure for SLOAV and then introduce the two-phase message transmission technique. We then explain how data elements are rotated, and transmitted in the SLOAV algorithm.

New Data structure for SLOAV Algorithm The new data structure support for SLOAV is shown in Fig. 4.1. Suppose there are 5 processes. For each process in the AlltoallV communication, there is an input data buffer that contains N data elements of various sizes ready to send to N processes respectively. The input data buffer is enough for a linear communication algorithm. To enable a logarithmic algorithm for AlltoallV, we add two new data structures: Element Index Table (EIT) and SLOAV buffer.

The EIT includes N entries, each of which is for a single data element. Every entry contains a length specifying the size of element and an offset pointing the location of the element in the input buffer or SLOAV buffer. Three benefits are obtained by using the EIT structure. First, each element is not required to be of the same length, and data can be placed anywhere, thus providing high flexibility. Second, it can significantly avoid memory copy overhead, because we only need to modify the length and pointer in the EIT without shifting actual data when a data movement is required. Compared to Bruck algorithm, which requires local memory rotation and data movement in both Step 1 and 3, EIT can effectively reduce the amount of memory copy. Thirdly, this structure is space-efficient and provides fast search speed.

SLOAV buffer is a temporary buffer to hold the intermediate elements that are too large to fit in its element segment. The SLOAV buffer is acquired from the Cheetah buffer pool (allocated to each process during MPI_INIT). Its size is adjustable, depending on the number of processes and average element size.

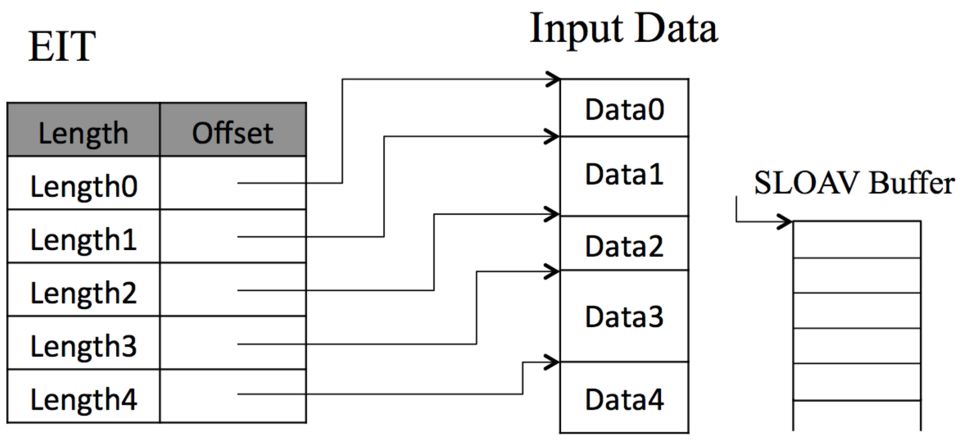


Figure 4.1: Data structure for SLOAV

Two-phase message transmission A logarithmic algorithm for AlltoallV operation requires data transmission to intermediate processes. But intermediate processes have no knowledge of the exact size of intermediate data they will receive. Each process only knows how much data it needs to send out and the length of each data element to receive. Precalculating the sizes of data elements at each round requires significant extra collective operations, which can cause extra communication and computation overhead. In order to address this, we propose an effective two-phase transmission approach.

In this approach, a process combines the elements that have the same destination process id, then send the total message size and data to the targeted process. As shown in Fig. 4.2, three rounds of communications are required for an AlltoallV operation among five processes. In round

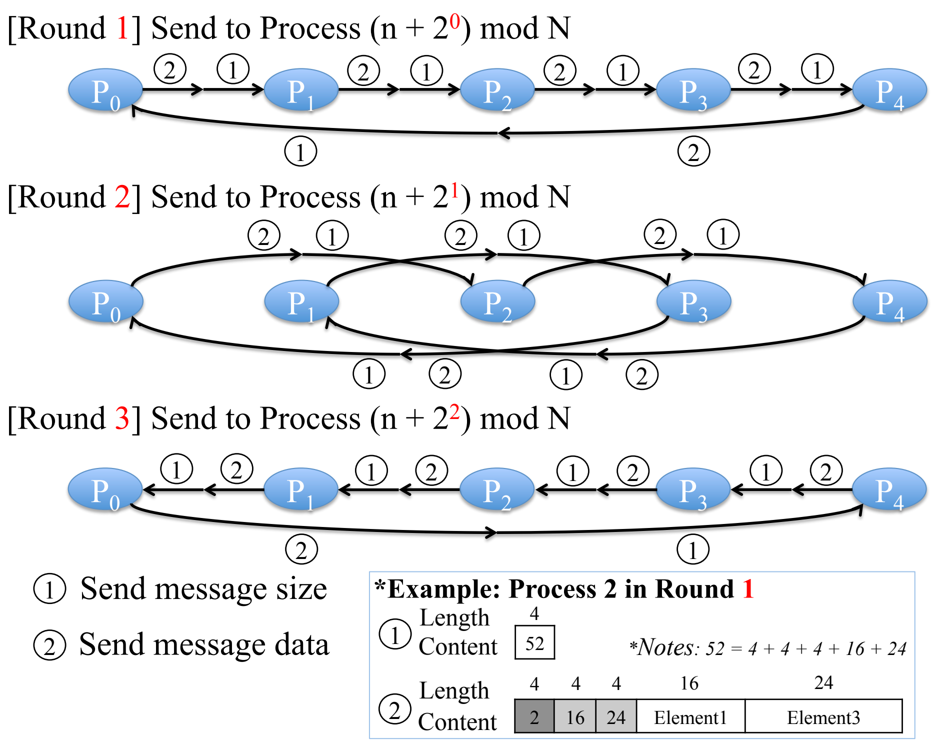
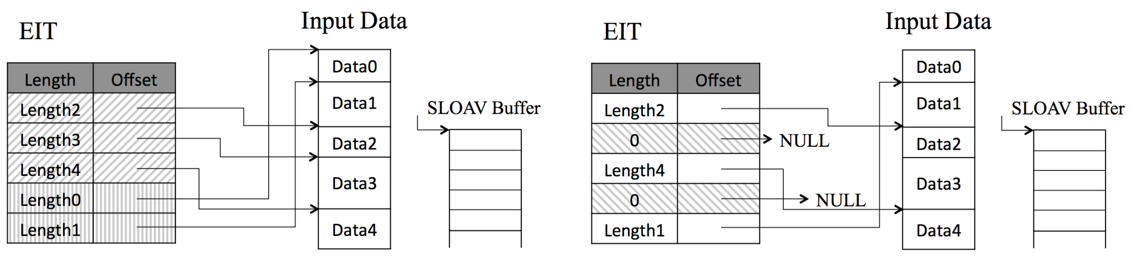
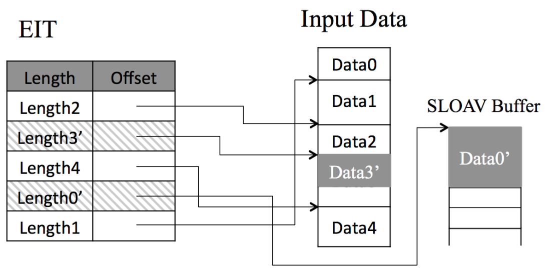

[Round 1] Send to Process (n + 2^0) mod N
P0
P1
P2
P3
P4
[Round 2] Send to Process (n + 2^1) mod N
[Round 3] Send to Process (n + 2^2) mod N
① Send message size
② Send message data
*Example: Process 2 in Round 1
Length
Content
52
*Notes: 52 = 4 + 4 + 4 + 16 + 24
Element1
Element3

Figure 4.2: SLOAV's Process to Process Communication

s, process n firstly merges all of the elements e with $(e/2^{s-1})\,mod\,2$ equal to 1, then sends the size and data of the merged message to process $(n+2^{s-1})\,mod\,N$. Similarly, it receives the size and data of a combined message from process $(n-2^{s-1}+N)\,mod\,N$. In the example, for round 1, process 2 merges its element1 (16B) and element3 (24B) as a message, next, it sends the size of the message as a new message (4B) to process 3, and then send the merged (52B) to process 3. The merged message contains the number of elements, the lengths of each element and the data. At the same time, process 2 receives the message size and data from process 1. Totally, the two-phase message transmission requires $2 * log_2 N$ start-up costs for each process but is still orders of magnitudes faster than linear algorithm which takes N start-up costs, especially for communication among large number of processes and small sizes of messages.



(a) Local Rotation (b) Send Data to Another Process



(c) Receive Data from Another Process

Figure 4.3: Operations of each process in *SLOAV* algorithm

Element rotation, send and receive in SLOAV Fig. 4.3 illustrates how one process rotates, sends and receives the message elements in SLOAV for AlltoallV communication among five processes.

Fig. 4.3(a) depicts the status of a process after it rotates by two positions. Note that for local data rotation in one process, we only need to rotate the entries in EIT without data moving.

As shown in Fig. 4.3(b), after some elements are sent out, the value of that element entry in the table is set to NULL. Before a process receives new data element, the former data element at the same position of input data buffer must be sent out as required by the algorithm to avoid conflicts.

In Fig. 4.3(c), when an element arrives, if the length of received element is smaller than the capacity of the element segment in the data input buffer, the received data element is placed into the data input buffer like data element Data3. Otherwise, that element is placed into SLOAV buffer such as Data0 and the offset of that entry is set to point to its location in the SLOAV buffer. We use a unified range for the offset value in the EIT, if the offset value *i* is smaller than the size of the input buffer S, it directly points to the position of i in the input data buffer. If not, it points to position of $i-S$ in the SLOAV buffer. This SLOAV buffer can be recycled if the occupying data element has been sent out later.

4.1.1.2 Theoretical Analysis of Communication Complexity

During the AlltoallV collective communication, the size of data sent by one process to the other is unknown to the receiver. Therefore, it is imperative for the sender to firstly notify the receiver about the size of message before conducting the data transportation. This educes the basic model for a process to accomplish data transfer in AlltoallV operations:

$$T_{SLOAV} = T_{size} + T_{msg} \tag{4.1}$$

where T_{size} is the time for the sender to notify the receiver about the size, and the T_{msg} is the time to send message data. From the Hockney model [**?**], assuming there's no contention in the network, the latency of sending data between a pair of end-points can be modeled as

$$T = S + \frac{c}{B} \tag{4.2}$$

, where S stands for the cost of start-up, B is the network bandwidth, and c the amount of data transferred between endpoints. Based on this point to point exchange formula in which the time required to perform local memory copy is ignored, the time of transmitting message size and data can be expressed by the following two equations:

$$T_{size} = \lceil log_2 N \rceil * (S + \frac{4}{B}) \tag{4.3}$$

$$T_{msg} = \lceil log_2 N \rceil * S + \frac{\sum_{i=0}^{\lceil log_2 N \rceil - 1} \vec{C}_k * \vec{M}_{\lceil log_2 N \rceil - 1 - i}}{B} \tag{4.4}$$

In the two formulas above, N is the total number of processes in the communication. $\lceil log_2 N \rceil$ rounds are required to complete process to process communication. A 4-byte *Int* variable is large enough to indicate the size of message. The time used in each round to send the message size is the sum of start-up time and the 4-byte integer tranmission time.

To calculate the message latency, besides the cost of start-up, the time of sending all of the data by each process in the network needs to be calculated. Equation (4.4) is the latency of adopting SLOAV to transfer data elements of various lengths. Vector $\vec{C_k}$ is a $1 \times N$ array, storing the length of process k's each data element that needs to be sent to N processes (including itself) in the current round. Thus $\vec{C_k}$ changes at every round. M is a $N \times \lceil log_2 N \rceil$ binary matrix, $\vec{M_k}$ is M's k-th column, and the combination of 1 or 0 at each row is the binary number of the corresponding row index. For example, when N is 5, the M is shown in Equation 4.6. M indicates which elements the sender needs to send at each round, starting from right to left.

$$\vec{C_k} = \begin{bmatrix} c_{k0} & c_{k1} & \cdots & c_{k(N-1)} \end{bmatrix} \tag{4.5}$$

$$M_{5*\lceil log_2 5 \rceil} = \begin{bmatrix} \vec{M_0} & \vec{M_1} & \vec{M_2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{4.6}$$

In contrast, the cost of linear AlltoallV algorithm is:

$$T_{linear} = N * S + \frac{\sum_{j=0}^{N-1} c_{kj}}{B} \tag{4.7}$$

As can be seen from the above formulas, when the average message size is small, the collective communication is dominated by the overhead of start-up operation (S). So the logarithmic algorithm can achieve better performance, which only requires $\lceil 2 * log_2 N \rceil$ times of start-up costs. However, when the message size becomes large, the network transmission time turns to be bottleneck. In addition, because in logarithmic AlltoallV, intermediate processes need to transfer or relay the messages, it can cause lots of duplicated messages sent/received in the network. For that reason, the performance of SLOAV is not ideal for dealing with large messages. If the maximum amount of data needs to be received in the AlltoallV operation is greater than a threshold, the program will switch to the large message algorithm.

4.1.2 SLOAVX: Optimize SLOAV on Multicore Systems

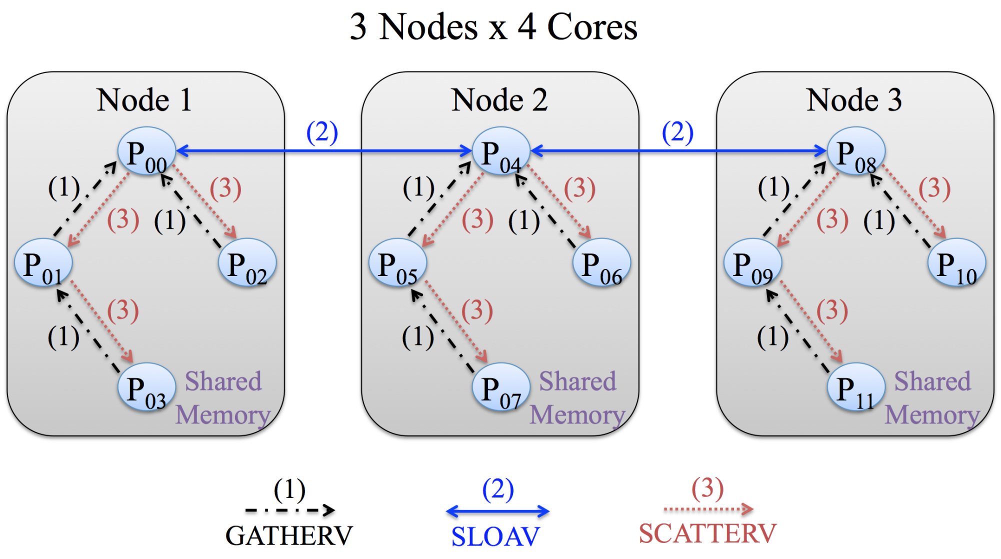


Figure 4.4: *SLOAVx* Algorithm

We have described the *SLOAV* algorithm, which supports AlltoallV to realize a global message exchange in a logarithmic number of rounds. Although *SLOAV* reduces the complexity, the required number of messages to achieve AlltoallV is still a function of the number of processes involved in the communication. Nowadays, large-scale scientific applications launch hundreds of thousands of processes to fulfill task completion. At such scale, message latency can quickly prevents *SLOAV* from achieving the optimal performance. Therefore, in this section, we exploit the advantage of fast shared memory in multicore systems to optimize the *SLOAV* algorithm. We name such optimization as *SLOAVx*, which collects all messages from all processes on the same node before conducting inter-node communication. As a result, *SLOAVx* can efficiently cut down the number of messages exchanged between nodes, reducing it to be a function of the number of nodes in the cluster, thus dramatically mitigating the performance impact of network message latency.

Different from SLOAV algorithm which is unaware of the topology of processes, *SLOAVx* collects all messages from all local processors and delegates the group leader to perform inter-node communication via the *SLOAV* algorithm. Fig. 4.4 illustrates the high-level description of *SLOAVx* algorithm. On each node, one process is selected as the group leader. Such leader election can be achieved through using simple consensus algorithm. For the example in Fig. 4.4, processes P_{00}, P_{04}, and P_{08} are selected as group leaders for the processes on node 1, 2 and 3, respectively. After the leader is selected, all the rest processes on the same node are organized into a binomial tree, as shown in the Fig. 4.4. Such organization prevents the root process (group leader) from being the potential bottleneck and allows multiple processes to collect and forward messages simultaneously. Note that, under such tree structure, each process is aware of its parent and children processes based on the topology information provided by *SBGP* component in Cheetah framework.

Upon finishing the above setting up phase, on each node, *SLOAVx* continues AlltoallV communication by determining the offset for each process in the shared memory into which each process writes the message data. To calculate such offset, each process needs to notify its parent about the message size at the first place and then write the actual data. This step is shown in Step (1) in the Fig. 4.4. However, there is no existing MPI function support such functionality. Therefore, *SLOAVx* modifies *MPI_Gatherv* function to achieve such local message gathering in the shared memory.

Once all messages from all local processes have been collected, the group leader partitions the data according to the destination processes, and groups all the data going to the same compute node into the same partition. In our example, on each node, there are three such partitions, each one of which targets at a different compute node. After that, group leaders of the compute nodes apply the *SLOAV* algorithm to exchange the partitions, as shown in Step (2) in Fig. 4.4.

Upon receiving the data partitions for all the processes on the node, a group leader needs to transmit the data to each local process. Similar to *gatherv* in Step (1), such functionality is achieved by asking processes to read data from shared memory. Therefore, in Step (3), group leader notifies each process about the offsets from which each local process can read data from all the other processes in the cluster. Such notification goes through the tree structure instead of using broadcast. *SLOAVx* employs a modified *MPI_Scatterv* function to achieve this.

4.2 Design and Implementation of JVM-Bypass Shuffling

In this section, we are going to explain the design and implementation of our JVM-Bypass Shuffling in Hadoop MapReduce.

4.2.1 Design of JVM-Bypass Shuffling of Intermediate Data

We firstly describe the software architecture of JVM-Bypass Shuffling and then several salient features of its internal design.

4.2.1.1 Architecture of JVM-Bypass Shuffling

To address the issues discussed in section 3.2.1, a shuffling library, called JVM-Bypass Shuffling, is proposed for moving Hadoop intermediate data. Figure 4.5 shows the changes in the software architecture from the original Hadoop to JBS. The main objective of JBS is to avoid JVM's heavy overhead caused by its deep stack of transport protocols without changing the user programming interfaces such as the user-defined *map* and *reduce* functions. Instead of going through a stack of Java HTTP and socket libraries (shown at the bottom left of Figure 4.5), JBS is designed to bypass the JVM from the critical path of the intermediate data shuffling.

Additionally, to expand the network portability of Hadoop and compensate its lack of RDMA support, we have designed JBS as a portable layer on top of any network transport protocol. As shown in the figure, both RDMA and TCP/IP protocols are integrated as the underlying network mechanisms for data transfer in JBS. As a result, the JBS library is designed to avoid the overhead of JVM on data shuffling and accelerate Hadoop on two most popular commodity cluster networks including Ethernet (1Gigabit Ethernet and 10Gigabit Ethernet) and InfiniBand.

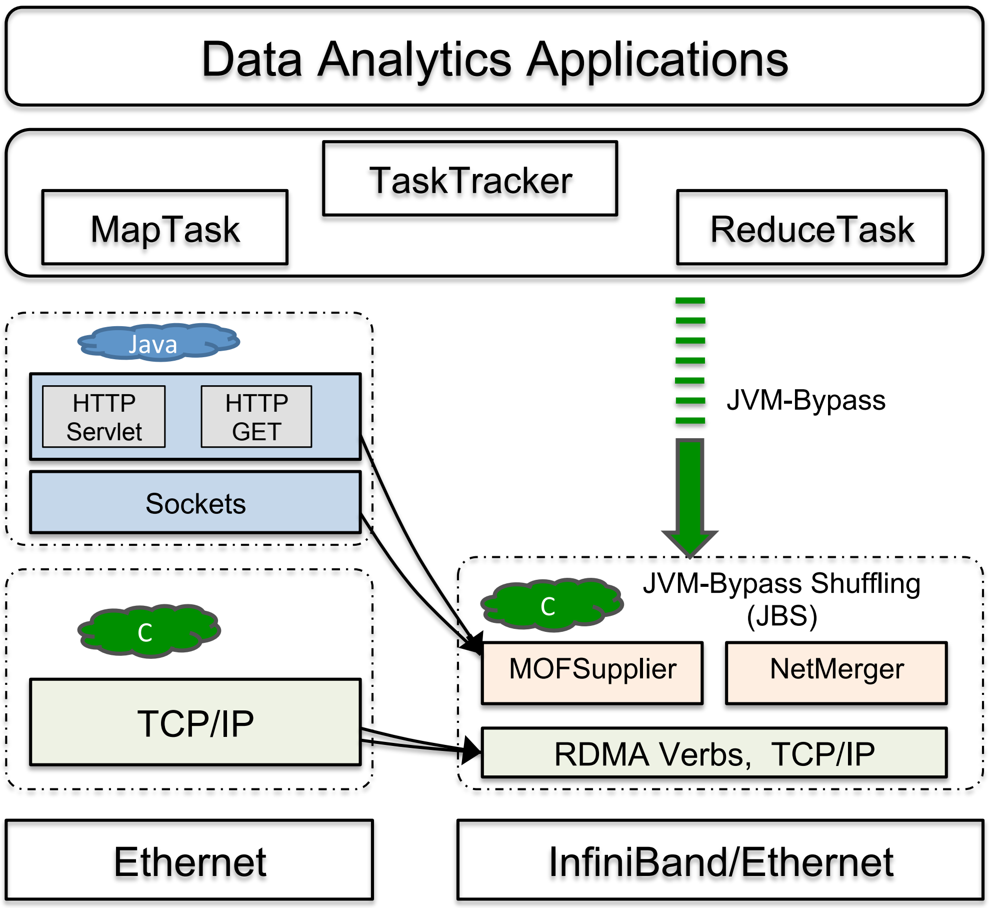


Figure 4.5: Software Architecture of JBS

JBS as a Transparent Plugin Library – While the main objective of JBS is to bypass JVM, it is also important to design it as a transparent plugin library to Hadoop. In doing so, not only Hadoop programs can leverage JBS without any change, but also JBS can work with other Hadoop internal components as a plugin module. To this end, two components, namely MOFSupplier and NetMerger, are introduced to undertake the movement of intermediate data for Hadoop. These two components are standalone native C processes. They are launched by the local TaskTracker, with which they communicate via loopback sockets. These two components replace the HttpServlets in the TaskTracker and the MOFCopiers in the ReduceTasks, respectively, thereby bypassing the need of JVM when moving data between MOFSupplier and NetMerger. As a plugin module [9] for Hadoop MapReduce, JBS is invoked based on a runtime user parameter. When it is not loaded, it does not change the execution of the original Hadoop.

4.2.1.2 Pipelined Segment Prefetching

In the original Hadoop, for each fetch request, the HttpServlet finds the MOF and its corresponding Index file from disk devices. It then retrieves the location of the targeted segment in the MOF based on the Index file. Note that an *IndexCache* is usually maintained to cache the entries from the Index file and speed up the identification of MOF segments. From the MOF, a segment is read via disk I/O and then transmitted through network I/O. As shown in Figure 4.6, disk read and network transmit (*Xmit*) are completely serialized in the HttpServlet. No effort is made to correlate and batch multiple requests to improve the locality of disk accesses. As a result, each request can experience a long delay in the queue, degrading the performance of data shuffling.

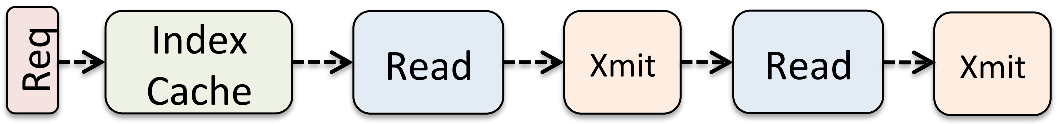


Figure 4.6: Serialized Request Processing in HttpServlet

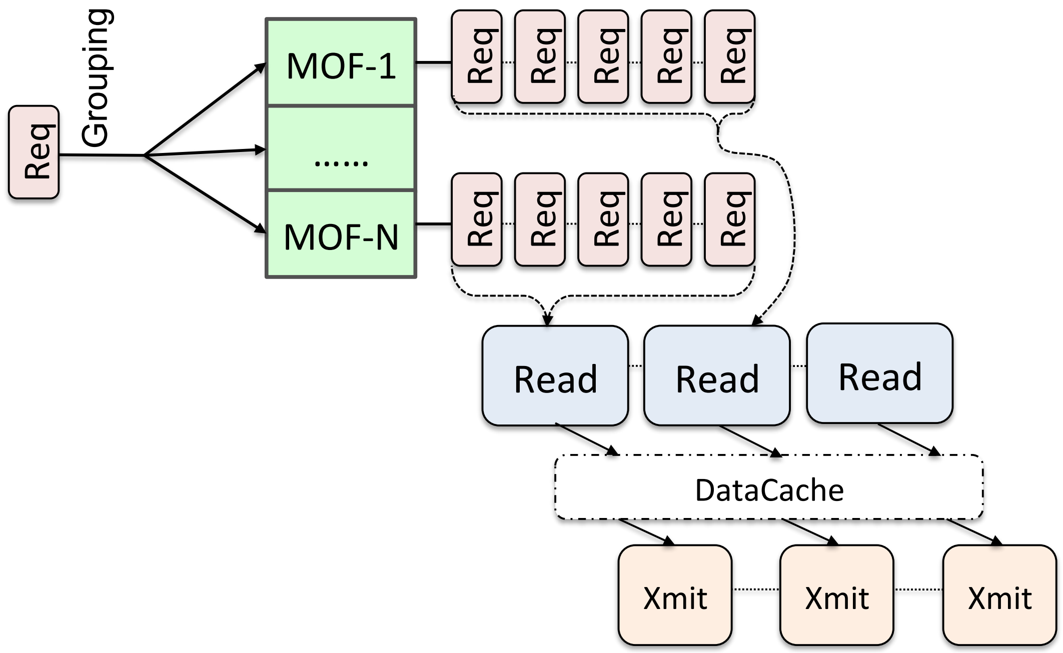


Figure 4.7: Pipelined Segment Prefetching in MOFSupplier

To improve the efficiency of serving intermediate data, we design the MOFSupplier with a pipelined prefetching scheme. Besides providing an IndexCache for quick identification of MOF segments, we also design a *DataCache* to prefetch MOF segments for request processing. As shown at the bottom of the Figure 4.7, with dedicated memory space as the DataCache, requests are grouped based on their targeted MOF, and those in the same group are ordered based on their intended segments. Segments for several requests are prefetched to the DataCache. All groups are served by the disk prefetch server in a round-robin manner. When a batch of segments are ready in the DataCache, they are transmitted over by asynchronous network operations. With the DataCache as the buffer, fetch requests are served in a batched and pipelined manner, thereby increasing the locality of disk accesses and reducing the average delay of requests.

4.2.1.3 Consolidated and Balanced Data Fetching in NetMerger

As mentioned earlier, a Hadoop ReduceTask employs multiple MOFCopier threads to concurrently fetch independent segment data to local file system. Several merging threads are running in the background to merge available segments. When faced with large data sets, both MOFCopier and merging threads spill data to local disks. As part of our Hadoop Acceleration project, the NetMerger for JBS is designed to undertake the shuffling (a.k.a fetching) and merging of intermediate data. The details of the Network-Levitated Merging algorithm used by NetMerger to conduct the segments merging is discussed in our previous paper [61]. Here we provide more details on the arrangement of network requests to fetch data from remote MOFs.

As shown in figure 4.8, we design the NetMerger as a component that can consolidate network fetching requests from all ReduceTasks on a single node. As mentioned in Section 4.2.1.1, only one NetMerger is created by TaskTracker on the single node. Thus all segments needed by multiple ReduceTasks on the same node will be served by the NetMerger. Within this shared NetMerger, we consolidate and group all requests based on their targeted remote nodes. Requests to the same node are ordered based on their time of arrival. Using this organization, we are able

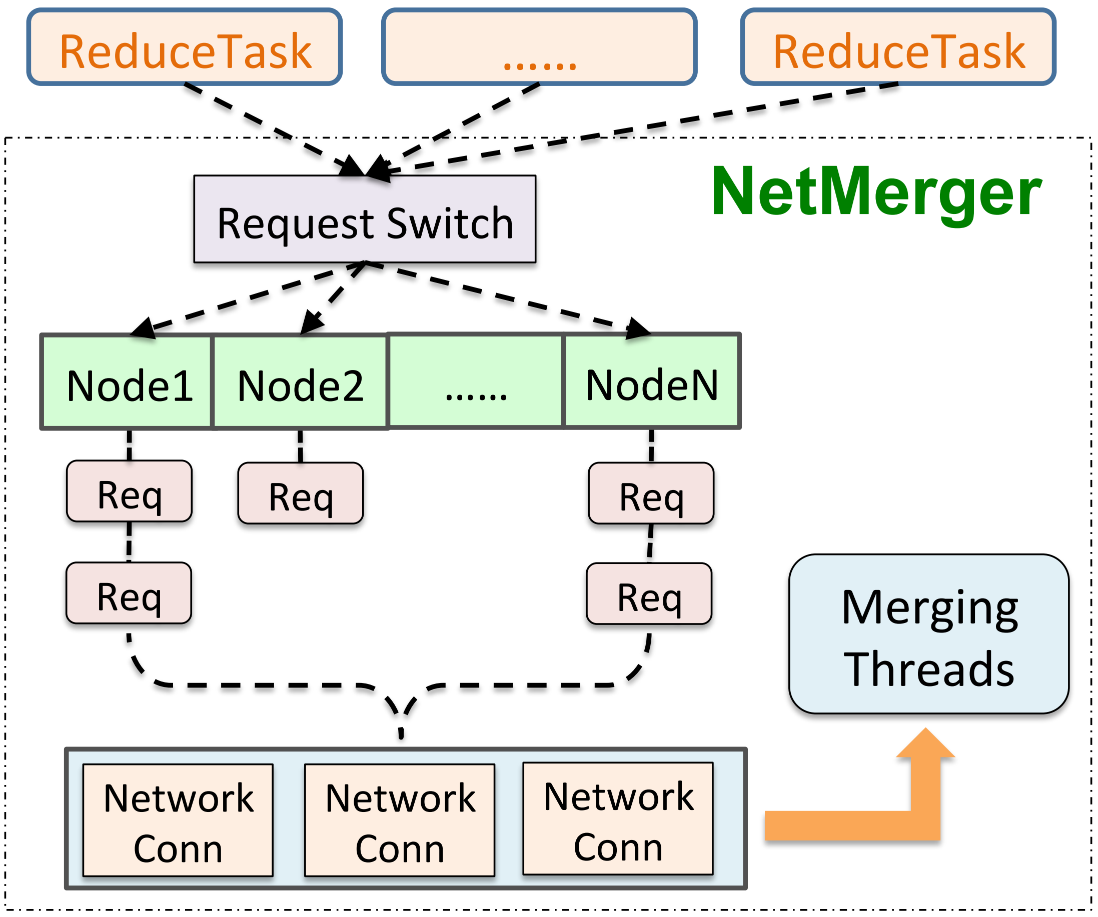


Figure 4.8: Consolidating Fetching and Connections in NetMerger

to consolidate a number of network connections, which is no longer the total amount of MOF-Copiers from all ReduceTasks. This consolidation also reduces the resource requirements from creating and sustaining many network channels and their associated memory to buffer data. In addition, across different groups, we adopt a simple round-robin mechanism to balance the injection of fetching requests to different nodes, mitigating the impact of burst requests from an aggressive ReduceTask.

4.2.2 Implementation

We have implemented JBS as a portable library for different network environments including the traditional TCP/IP protocol, the RDMA protocol and the latest RoCE (RDMA over Converged

Ethernet) protocol. The implementation of JBS is same for both RDMA and RoCE, except that their activation is different. In this paper, we refer to RDMA as the protocol activated on InfiniBand and RoCE as the protocol activated on 10Gigabit Ethernet.

The TCP/IP protocol and the RDMA-like (RDMA and RoCE) protocols are very different in their ways of establishing network connections. Accordingly, we provide some implementation details for their respective connection establishment.

4.2.2.1 Connection Establishment for RDMA and RoCE

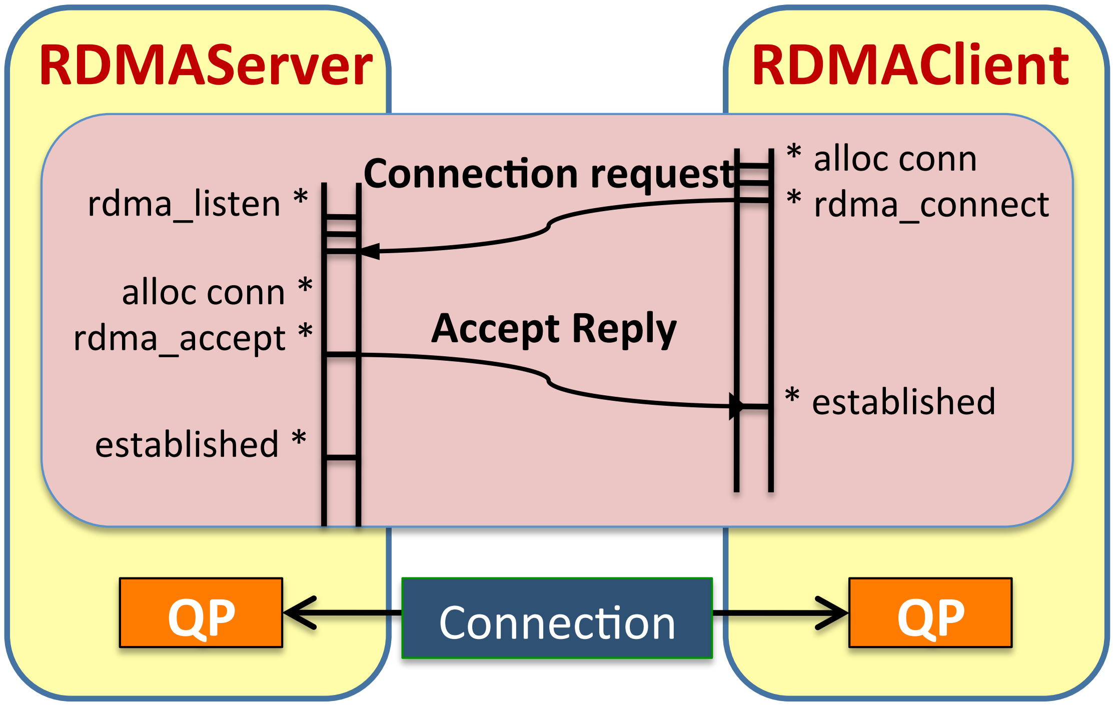


Figure 4.9: Connection Establishment for RDMA and RoCE

Figure 4.9 illustrates our implementation of connection establishment on RDMA and RoCE. A pair of RDMAServer and RDMAClient are designed to handle RDMA connection establishment. An additional thread managing network events is created for both RDMAServer and RDMAClient. The first fetching request triggers a RDMAClient to initiate the process of connection establishment with the remote RDMAServer. In the case of a RDMAClient, it allocates a new connection (a.k.a Queue Pair) and sends a connection request via *rdma_connect()* to the RDMAServer.

The network thread listening for incoming requests on the RDMAServer receives this connection request and handles a series of events that are detected on the associated RDMA event channel. This RDMAServer then allocates a new RDMA connection. Via *rdma_accept()*, it accepts and confirms the connection request to the RDMAClient. The successful completion of the *accept()* call will be detected via an *established* event by network threads at both RDMAServer and RDMAClient. This completes the establishment of a queue pair (QP), i.e., a new RDMA connection.

Currently we use only the Reliable Connection (RC) service provided by RDMA-capable interconnects. Since the cost of setting up RDMA connection is relatively high, we keep newly created connections for reuse by default. We allow a maximum of 512 active connections. When this threshold is reached, connections are torn down based on the LRU (Least Recently Used) order.

4.2.2.2 TCP/IP-Based Communication

To support the TCP/IP protocol, JBS employs conventional TCP/IP sockets to establish connections. It makes use of an event-driven model and multiple threads to achieve good parallelism and communication throughput. On the client side, one thread is dedicated to prepare connection requests to different nodes and monitor their status. The actual connection requests are made by the client's data threads to the remote servers. On the server side, one thread is listening for client connection requests. It accepts a client's connection request after validating its legitimacy. Both client and server use the *epoll* interface to monitor and detect events from concurrent connections, and rely on their data threads to perform the network communication for data transfer.

Different from RDMA protocol, since the cost of setting up and tearing down TCP/IP network connection is limited, a client's TCP/IP fetch request triggers the creation of a TCP connection to a remote node on demand. We do not keep the TCP connection, a connection is torn down when the node to node data fetching procedure is accomplished.

4.3 Developing an Efficient Virtualized Analytics Shipping Framework

To tackle the aforementioned challenges, we propose a Virtualized Analytics Shipping (*VAS*) framework in this paper. As depicted in Fig. 4.10, we leverage KVM [5] to segregate MapReduce jobs from the I/O services running on OSS nodes, through which the performance stability and system reliability of Lustre storage servers can be guaranteed. To be specific, each OSS hosts one virtual machine (VM), which installed YARN and Lustre client related drivers to interact with Lustre file system. Thus the collection of KVMs on all OSSs form a virtualized cluster, to which the analytics programs are shipped.

Simply porting original data-intensive programming model-YARN into computing-intensive HPC environment to work with Lustre file system can only deliver sub-optimal performance due to distinct features of Lustre and HDFS [40]. We develop three main techniques in VAS to achieve efficient analytics shipping. First, we optimize the performance of network and disk I/O via dynamic routing configuration of VMs. Second, we develop stripe-aligned data distribution and task scheduling for speeding up data accesses in MapReduce. Finally, we propose a new technique that can avoid the explicit shuffling and pipeline the merging and reducing of intermediate data in our VAS framework.

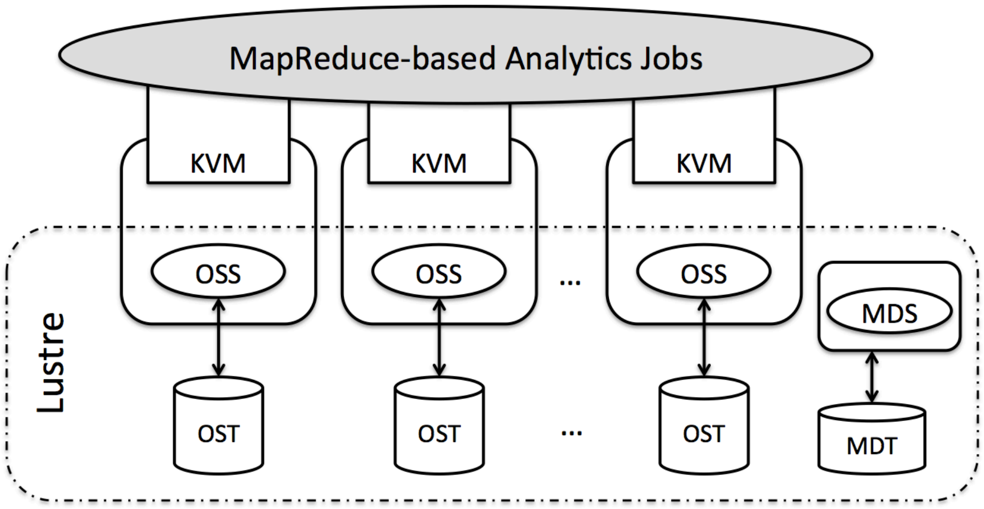


Figure 4.10: MapReduce-based Analytics Shipping Model

4.3.1 Fast Network and Disk I/O

While we utilize KVM for segregating MapReduce from I/O services on Lustre servers, a major concern is the need of appropriate virtualization techniques for delivering efficient network and disk I/O to the VMs.

4.3.1.1 Network virtualization

There are two widely used advanced I/O options in KVM: VIRTIO and single root I/O virtualization (SR-IOV). VIRTIO is a virtualization standard for network and disk device drivers. It performs much better than “full virtualization” because VIRTIO does not require the hypervisor to emulate actual physical devices. SR-IOV provides VMs with direct access to network hardware resources, eliminating the overhead of extra memory copies. A major difference between VIRTIO and SR-IOV network configuration is the use of switch-bypassing: in VIRTIO, a bridge is created for the VM-to-host communication without going through the switch; in SR-IOV, all communication has to go through the switch, even when a VM communicates with its host.

To obtain optimal network performance, we have firstly evaluated both technologies for three different connection scenarios. By leveraging iperf benchmark, we test the point to point bandwidth of each connection scenario for VIRTIO and SR-IOV. The results are showed in Table 4.1.

The results of Cases 1 and 2 reveal that VIRTIO bridge networking can achieve about 15.2 Gbps bandwidth, which is much faster than SR-IOV method because of VIRTIO’s bypassing of the switch.

However, SR-IOV outperforms VIRTIO in terms of remote network connection. For example, in Case 4, SR-IOV is able to achieve 6.1 Gbps bandwidth for the connection between the VM and the remote Physical Machine (PM), and 9.2 Gbps for remote VM-to-VM communication (Case 6). This is close to the 10 GbE line rate. Such performance differences are because the software-based approach of VIRTIO requires redundant memory copies for sending/receiving packages in the host. In contrast, SR-IOV allows the direct exchange of data between VMs and thus avoids extra memory copies.

Since VIRTIO and SR-IOV can achieve superior performance for the local VM-to-PM (Case 1) and remote VM-to-VM (Case 6) scenarios, respectively, we adopt a dynamic routing policy for network traffic of VMs. To implement this policy, we firstly ensure that the routing table of both VM and PM are set up appropriately. By default, the routing scheme uses SR-IOV. If the routing scheme detects that the source and destination reside on the same node, it switches to VIRTIO.

Case #	Source	Destination	Interface	BW (Gbps)
1	hostA-VM	hostA	VIRTIO	15.2
2	hostA-VM	hostA	SR-IOV	7.1
3	hostA-VM	hostB	VIRTIO	4.3
4	hostA-VM	hostB	SR-IOV	6.1
5	hostA-VM	hostB-VM	VIRTIO	7.2
6	hostA-VM	hostB-VM	SR-IOV	9.2

Table 4.1: Throughput Performance for Different Communication Cases

4.3.1.2 Fast Disk I/O in VMs with VIRTIO

For disk I/O, we apply VIRTIO for attaching storage devices to the KVM. To evaluate its effectiveness, we have conducted experiments to measure I/O performance of both the VM and PM. We use the IOzone benchmark. As shown in Fig. 4.11, the results indicate that VIRTIO achieves an I/O performance for the VMs that is comparable to that on PMs for various access patterns such as sequential reads/writes and random reads/writes.

4.3.1.3 KVM's I/O interference to Lustre storage servers

A major concern for the VAS framework is the performance interference to the I/O services of Lustre storage servers. To investigate this issue, we leverage IOzone benchmark to simulate heavy I/O workload and measure the aggregate I/O bandwidth of three cases: Lustre-alone, Lustre with a concurrent YARN program running on the physical machines (Lustre-YARNinPM), Lustre with a virtualized YARN program running on KVM virtual machines (Lustre-YARNinVM). For the last two cases, 100 YARN TeraSort jobs are launched one by one on 8 Lustre storage servers, at the

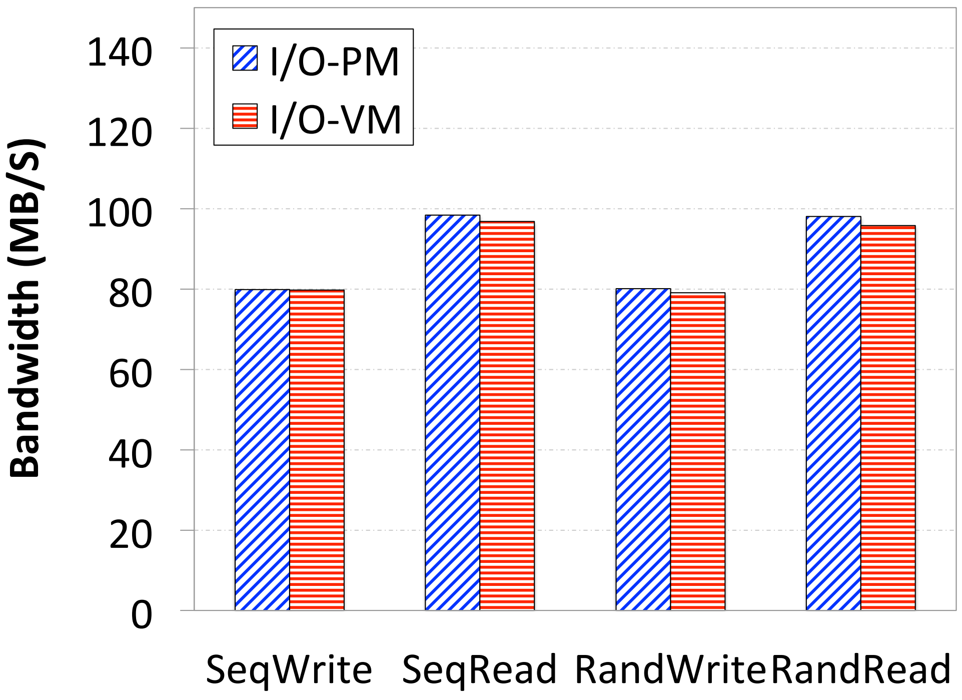


Figure 4.11: Disk I/O Performance of PM and VM

same time heavy I/O operations are conducted by IOzone Benchmark on remote Lustre clients to evaluate Lustre's Write/Read bandwidth.

As shown in Fig. 4.12, running YARN directly on PMs incurs a bandwidth degradation of 11.9% and 15.2% for sequential read and random read, respectively. In contrast, running YARN in VMs has negligible impact to the Lustre servers. In addition, since the Lustre write bandwidth is not sensitive to caching effect [52], the write bandwidths for all three cases are comparable.

4.3.2 Stripe-Aligned Data Distribution and Task Scheduling

The locality of task scheduling is an important strategy for MapReduce to minimize data movement during analytics. To exploit locality in VAS, we need to have a thorough understanding of its data movement. Fig. 4.13 shows the end-to-end path of data movement when running MapReduce on top of Lustre in our VAS framework. There are six main steps of data I/O, all happening on Lustre. A MapTask is involved in the first three steps: (1) initially reading the input

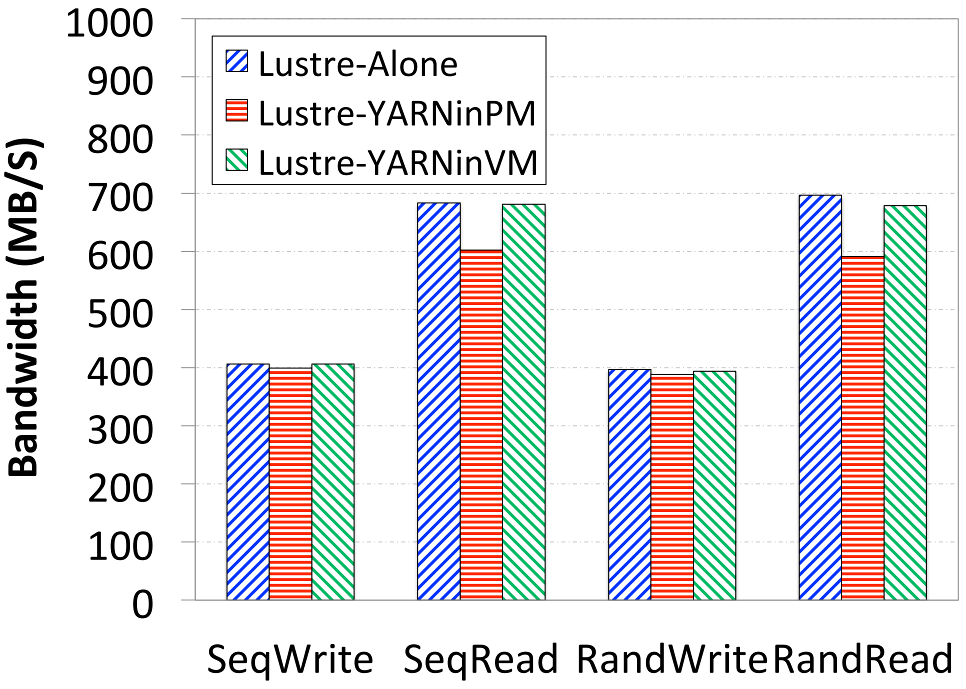


Figure 4.12: Using KVM to Alleviate YARN's Interference to Lustre

splits, (2) spilling data during the map phase, finally (3) merging and storing the intermediate data as a MOF on the Lustre. A ReduceTask is involved in the last three steps: (4) fetching and shuffling data from MOFs, (5) merging data segments and spilling some merged segments to Lustre as needed, and (6) reading spilled segments and writing final results at the end of reduce phase. A partial enhancement to this path may not result in an overall performance improvement, e.g., improving the locality in Step 1 as done in [40]. For an efficient analytics shipping framework, we need to provide a complete optimization of this end-to-end data path. In the rest of this subsection, we elaborate our strategies for data placement and task scheduling in Steps 1, 2, 3, and 6.

For both initial input datasets and temporary spilled data, we need to determine the placement and distribution on Lustre. On Lustre, it translates to a decision on the choice of stripe size and stripe width of MapReduce data files. Setting the stripe size smaller than the size of MapReduce input split will force a MapTask to read from many OSTs, resulting in the loss of locality and an increase of I/O interference. Using a stripe size larger than the split size can force multiple MapTask to read from the same OST, reducing the parallelism and increasing the lock contention.

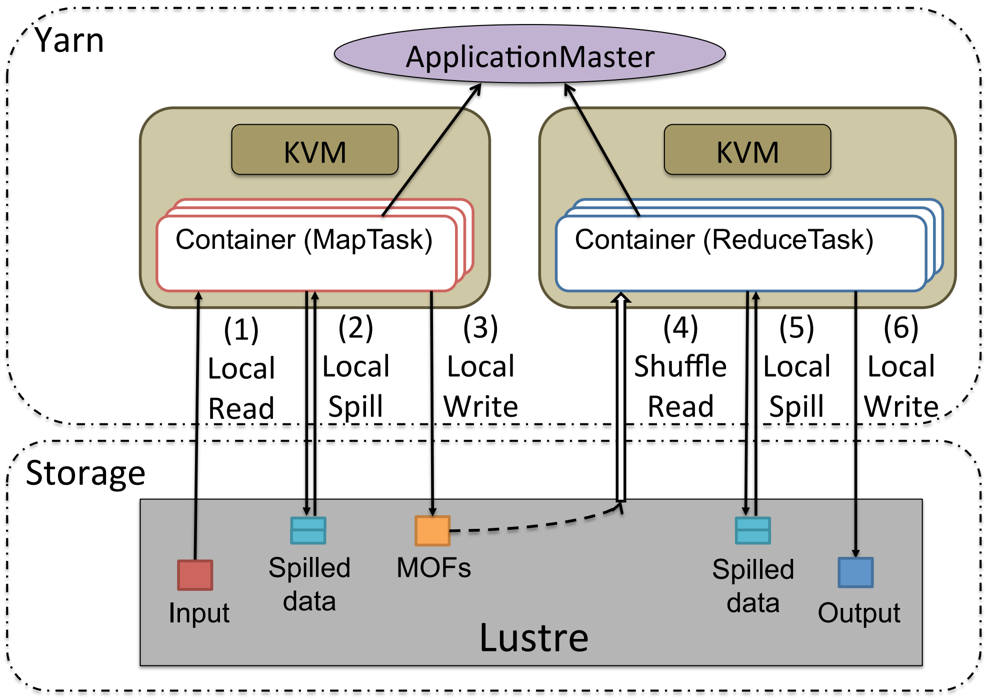


Figure 4.13: End-to-End Data Path for MapReduce On Lustre

Our tests indicate that it is beneficial to choose the Lustre stripe size equal to the input split size. We adopt this as the default configuration to distribute the stripes of MapReduce data. Note that Lustre records only the distribution of data stripes among physical machines, which cannot be directly translated to the actual VMs. To make the YARN scheduler aware of the correspondence of VMs to PMs, we create a configurable file with a mapping table of PMs and VMs. With this mapping table, we use the Lustre "lfs" tool inside the YARN MapReduce resource manager to distribute data in a stripe-aligned manner and place the initial input and the newly generated data in the end-to-end path (Steps 1, 2, 3, 5, 6) on the local OSTs. With the stripe-aligned distribution, in Step 1 of the end-to-end path, we enable the YARN scheduler to assign a MapTask preferentially to the storage server with an input split placed, i.e., striped on the local OST. The locality-based striping policy also helps Steps 2, 3, 5 and 6, in which the spilled data are temporarily written to the Lustre and soon read by the same task. Placing such data as a stripe on the local OST can reduce network and disk I/O to a remote OST. Steps 4 and 5 for the shuffling and merging of intermediate data are discussed next (see Section 4.3.3).

4.3.3 Pipelined Merging and Reducing of Intermediate Data

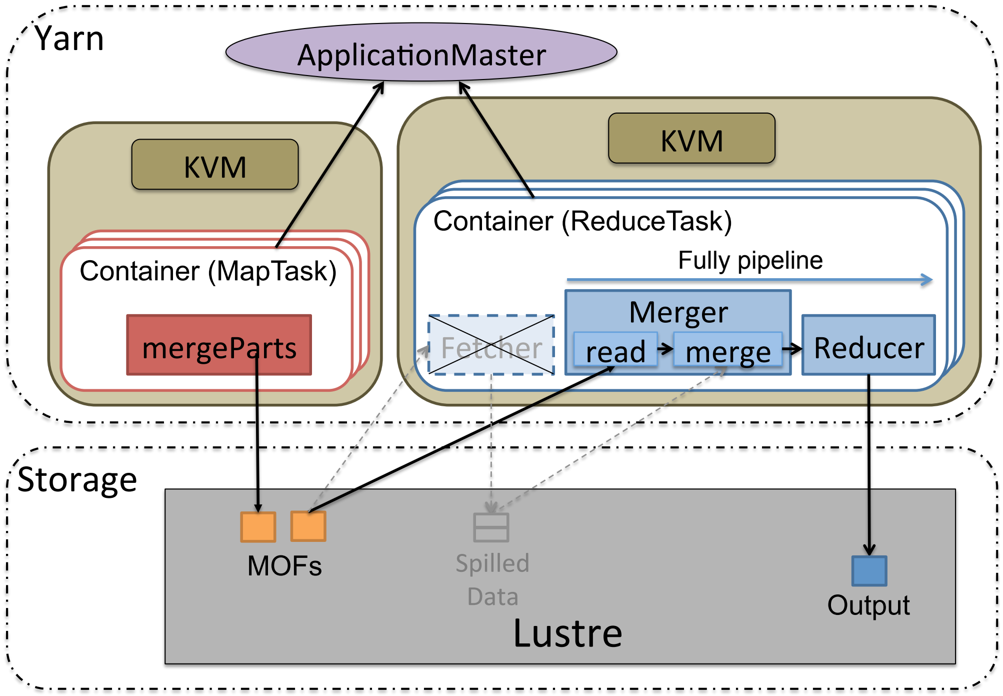


Figure 4.14: Pipelined Merging and Reducing of Intermediate Data without Explicit Shuffling

We take advantage of the fact that HPC storage systems provide a global shared file system such as Lustre. We are aware that some HPC storage systems such as Gordon [4] have solid state drives, which can break this assumption. However, we do not want to limit our analytics shipping framework to the environment with SSDs, and leave the exploration of SSDs in future work. We focus on exploiting the feasibility of analytics shipping in a typical Lustre environment.

With all the MOFs available to any task on Lustre, we can avoid an explicit shuffling stage in a ReduceTask. Even though there is still a need to bring the data into memory, merge them, and process them through the reduce function, a ReduceTask can go through an implicit shuffling step, i.e., reading data from MOFs through file I/O operations.

We describe our management of intermediate data based on the data path shown in Fig. 4.13. For Steps 4 and 5 in the end-to-end data path, we develop a technique called *pipelined merging and reducing of intermediate data* to mitigate the need of spilling and retrieving data. This technique fully leverages the fact that intermediate data stored as MOFs are available to all tasks without explicit shuffling. Specifically, we eliminate the explicit fetching step in a ReduceTask and form a new pipeline of data merging and reducing. As shown in Fig. 4.14, the optimized merger directly reads intermediate data generated by MapTasks from Lustre, and merges them into a stream of key-value pairs to the final reduce function. This new strategy can avoid the spilling of data in Step 4 and eliminate all the associated I/O operations.

To form a full pipeline of merging and reducing of data, the new merger does not start until all MOFs are available, similar to the strategy in [61]. In addition, we introduce two threads, one on file I/O and the other on merging data. These threads are pipelined to avoid the serialization of I/O and computation. Double buffers are provided in both threads to facilitate this pipeline. For example, in the file I/O thread, one buffer is provided for buffering data from Lustre, the other for buffering data ready to the merging thread.

Chapter 5

Evaluation Results

In this section, I show some evaluation results of the SLOAVx algorithm, Hadoop with JVM-Bypass Shuffling, and novel VAS framework.

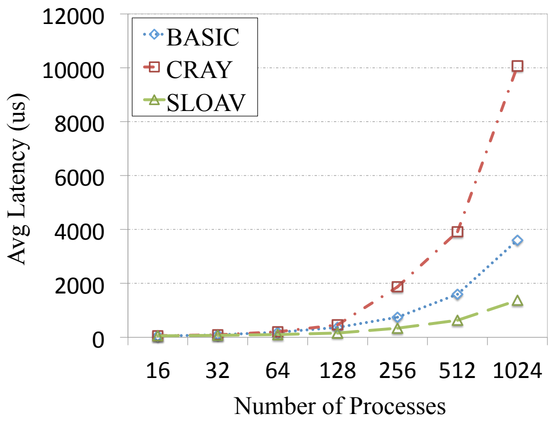
5.1 Performance Evaluation of SLOAVx

All experiments are conducted on the two environments, which are the Jaguar supercomputer and the Smoky development cluster at the Oak Ridge National Lab. Jaguar features 18,688 physical compute nodes, each with a 16-core 2.2GHz AMD Opteron 6274 processor and 32GB memory. All compute nodes are connected through Cray's high-performance Gemini networking system. The Smoky cluster contains 80 Linux nodes, each with 4 quad-core 2.0GHz AMD Opteron processors, 32GB of memory (2GB per core), and a gigabit Ethernet network, along with Infiniband interconnect. Both Jaguar and Smoky share a center-wide Lustre-based file system, called Spider.

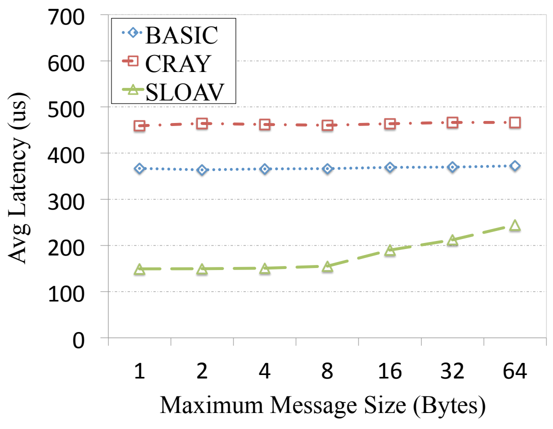
To measure the efficiency of SLOAV and SLOAVx algorithms, we employ the widely-used OSU Micro-Benchmarks 3.7 [44] and the popular NAS Parallel Benchmark (NPB) [7]. The OSU benchmark, uses MPI_Alltoallv operations in a tight loop to warm up the caches, and then measure the performance of 10,000 MPI_Alltoallv operations. The final result reports the average latency. Regarding to the NAS benchmark, we adopt IS (Integer Sort) kernel to evaluate the performance of MPI_Alltoallv function, every test runs IS for 10 times and reports the average. Note that our algorithm mainly focuses on optimizing the small message exchange, and is not very suitable for large data. Therefore, in our experiments, we focus on the evaluation of small messages.

5.1.1 Benefits of SLOAV Algorithm

We start our evaluation by measuring the latency of MPI_Alltoallv on the Jaguar supercomputer, and demonstrate the efficiency of SLOAV through two test cases. In the first test case, we increase the number of processes, while the message size varies from 1 byte to 8 bytes. In the second test case, we fix the number of processes (128) in the AlltoallV communication while varying the maximum size of message from 1 byte to 64 bytes. In both cases, we compare the results of SLOAV algorithm to those of the default MPI_Alltoallv function in Open MPI *(BASIC)* and MPI in Jaguar Cray system *(CRAY)*. We have also evaluated Tunned Open MPI [25], called *Tuned* MPI. However, across the tests, we observe that the performance of AlltoallV in Tuned MPI is very close to *BASIC* with 1%-3% difference. Therefore, in the following sections, we only report the results of BASIC for succinctness and clear presentation.



(a) AlltoallV with Maximum 8-bytes Small Message



(b) AlltoallV with Different Maximum Message Sizes

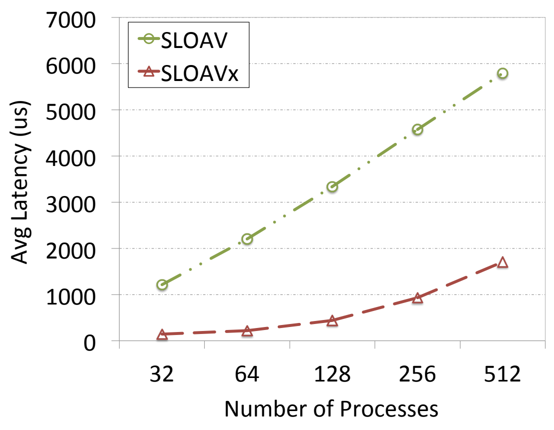
Figure 5.1: Effectiveness of *SLOAV* Algorithm

Fig. 5.1 (a) shows the results of first test case. On average, SLOAV algorithm significantly reduces the latency by up to 40.5% and 56.7%, when compared to the BASIC and the CRAY, respectively. More importantly, we observe that the improvement ratio increases proportionally to the number of processes involved in the AlltoallV communication, indicating superior scalability of SLOAV. For instance, when the number of processes increases to 1024, SLOAV outperforms

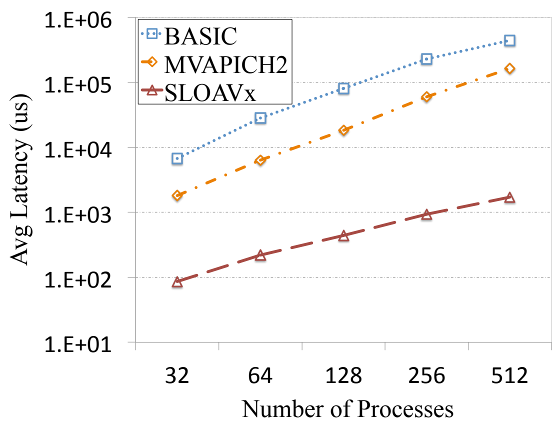
BASIC and CRAY by as much as 62.3% and 86.4% respectively. These results adequately prove the efficient design of our SLOAV algorithm.

The results of second test case are shown in Fig. 5.1 (b), in which the x-axis represents the maximum message size sent by one process to the other. As shown in the figure, SLOAV performs 51.4% and 61.4% better on average than BASIC and CRAY, and achieves up to 59.3% and 67.5% latency reduction when the maximum size is 1 byte. However, we notice that SLOAV only maintains a constant latency until the maximum message size reaches 8 bytes, after which, when message size further increases, SLOAV shows degraded performance. This is mainly because that in log scale algorithm all of the participant processes (including source and intermediate processes) need to merge different data elements that has the same destination into one message before sending them out, this can cause large amount of memory copies. In addition, the SLOAV algorithm can generate lots of duplicated data elements in the network due to message relay through intermediate processes, leading to larger size of the merged message in each communication round.

5.1.2 Benefits of SLOAVx Algorithm



(a) Comparison to SLOAV Algorithm



(b) Comparison to Other MPI_Alltoallv Solutions

Figure 5.2: Effectiveness of *SLOAVx* Algorithm

Recall from section 4.1.2 that *SLOAVx* algorithm is designed to optimize the SLOAV in clusters equipped with multi-core system by leveraging the advantages of shared memory on each node. Through aggregating all the messages from local processes before conducting inter-node communication, SLOAVx aims to reduce the number of network send/receive operations, thus improving the latency. In this section, we evaluate the performance of SLOAVx on the Smoky cluster and compare the results to that of SLOAV and other alternative MPI solutions. All the experiments in this subsection run 16 processes on each node.

We firstly compare the SLOAVx with SLOAV to assess the effectiveness of optimization. Fig. 5.2 (a) shows the comparison results. In the experiment, we increase the number of processes from 32 to 512, meanwhile using 16 bytes as the maximum message size. While both SLOAV and SLOAVx achieve logarithmic scaling trends, SLOAVx reduces the latency by 83.1% on average. We also observe a consistent improvement across all the tests. The improvement brought by SLOAVx is twofold. First, when running multiple processes on each node, SLOAV invokes traditional send/receive functions to conduct point-to-point communication even though they may reside on the same node. This can quickly lead to severe resource contention when the number of cores on each node increases, resulting in degraded system performance. In contrast, SLOAVx leverages fast shared memory operations, such as *gatherv* and *scatherv*, thus reducing the overhead of going through the network stacks. Secondly, SLOAVx aggregates all the messages from local processes before conducting the inter-node communication. This efficiently reduces the amount of communication occur on the network.

We further compare the SLOAVx with another two MPI alternatives, which are BASIC (described in section 5.1.1) and MVAPICH2 MPI. As shown in Fig. 5.2 (b), SLOAVx significantly reduces the latency by 99.3% and 97.4% on average when compared to BASIC and MVAPICH2, respectively.

5.1.3 Sensitivity on the Number of Cores Used on Each Node

As described above, SLOAVx strives to make the best use of shared memory on multi-core system. During the experiments, we observe that SLOAVx performs very differently when the number of cores used on each node differs. Fig. 5.3 presents such phenomenon. In the experiments, we carry out 3 test cases. In each case, we fix the number of total process in the AlltoallV communication, but change the number of cores used on each node. As shown in the figure, the more cores used on each node, the better performance SLOAVx is able to achieve. This effectively corroborates that SLOAVx can efficiently leverage the shared memory on each node to reduce the network operations.

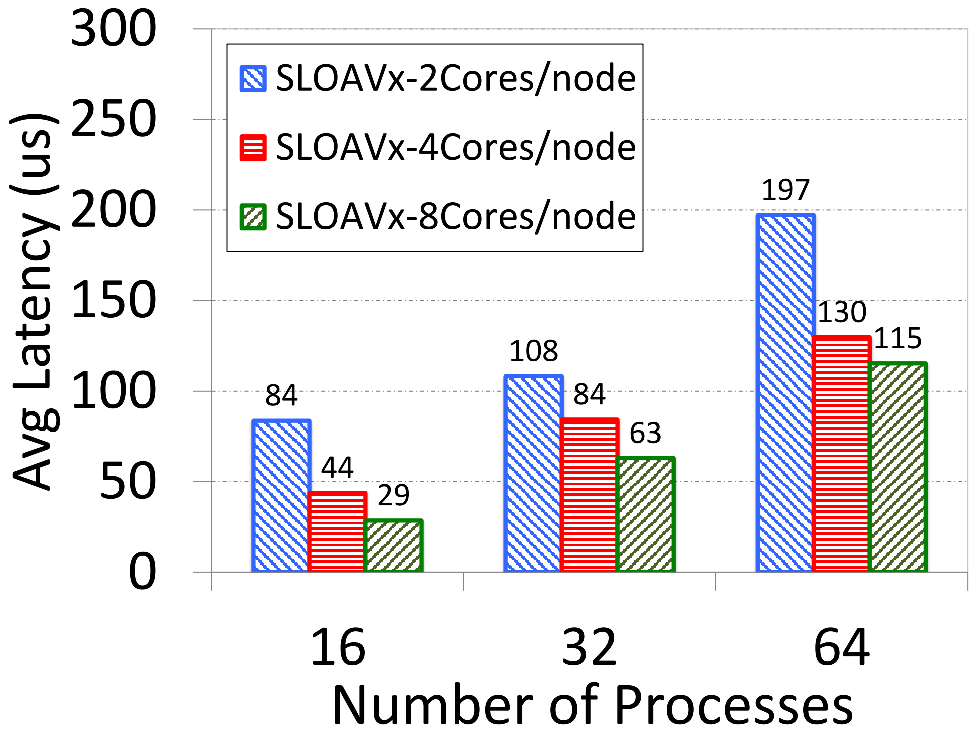
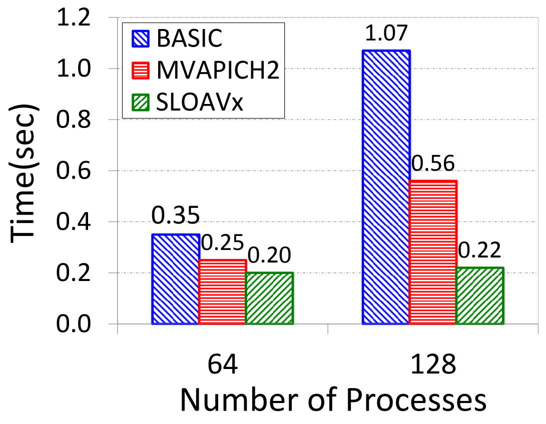


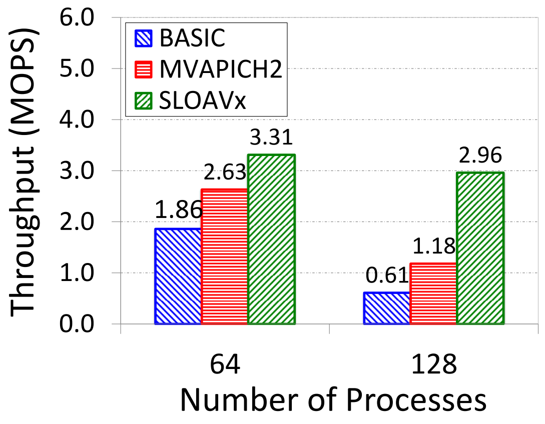
Figure 5.3: Effectiveness of *SLOAVx* with Different Cores/Node

5.1.4 Evaluation with NAS Parallel Benchmark

To further investigate the efficiency of SLOAVx algorithm for real-world applications. We adopt IS parallel kernel in NAS Parallel Benchmark. The IS kernel is derived from computation fluid dynamics (CFD) which applies MPI_Alltoallv to conduct integer sorting using bucket sort. It assesses both the job execution time and computation throughput (measured by MOPS). In this subsection, we conduct our experiments on *S* class and *W* class workload in IS kernel.



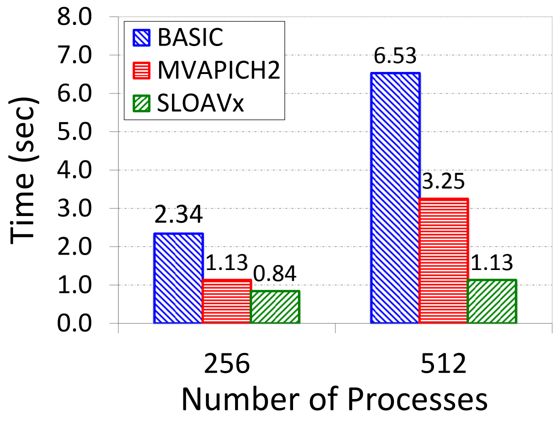
(a) Execution Time



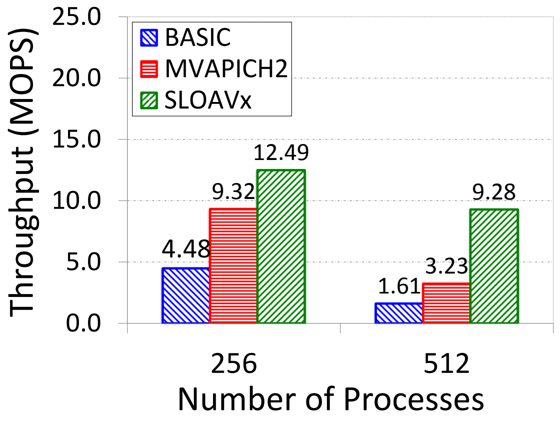
(b) Throughput

Figure 5.4: Improvement on *S* Class

Fig. 5.4 shows the results of running *S* class workload, which only supports up to 128 processes and sorts 64KB data size. As shown in the Fig. 5.4(a), SLOAVx can always achieve superior performance improvement over BASIC and MVAPICH2 regardless of the number of processes involved. The improvement can be as much as 79.4% and 60.7%, compared to BASIC and MVAPICH2, respectively when the number of processes is 128. In addition, Fig. 5.4(b) shows that SLOAVx can not only reduce the job execution time but more importantly increase the entire system throughput as well, reaching as much as 385.2% and 150.8% MOPS improvement when compared to BASIC and MVAPICH2.



(a) Execution Time



(b) Throughput

Figure 5.5: Improvement on *W* Class

The results of running *W* class workload are shown in Fig. 5.5. It aims to sort 1MB data and can support very large number of processes. As shown in Fig. 5.5(a) and 5.5(b), SLOAVx outperforms the other two alternatives in terms of execution time and throughput when the number of processes increases, and achieving as much as 476.4% and 187.3% throughput improvement ratio compared to BASIC and MVAPICH2 when running under 512 processes. Overall, these experimental results adequately demonstrate that SLOAVx can efficiently reduce the job execution time and increase the throughput.

5.2 Overall Performance of Hadoop with JVM-Bypass Shuffling

In this chapter, I present some initial experimental results to demonstrate the effectiveness of design of JVM-Bypass Shuffling and Network Levitated Merge. In sections that follow, I firstly show the high-level overall performance, then provide detailed performance analysis from different aspects.

All experiments are conducted on two clusters. Each cluster features 23 compute nodes. All compute nodes in both clusters are identical, each with four 2.67GHz hex-core Intel Xeon X5650 CPUs, two Western Digital SATA 500GB hard drives and 24GB memory. In the Ethernet environment, all compute nodes are interconnected by 1/10 Gigabit Ethernet. In the InfiniBand environment, all nodes are equipped with Mellanox ConnectX-2 QDR HCAs and connected to a 108-port InfiniBand QDR switch.

JBS can be adapted to various Hadoop versions. In our evaluation, we use the stable version Hadoop 0.20.3. During the experiments, one node is dedicated as both the NameNode of HDFS and the JobTracker of Hadoop MapReduce. The HDFS block size is chosen as 256MB to balance the parallelism and performance of MapTasks.

We run a group of benchmarks which include Terasort, WordCount, Grep from standard Hadoop package and SelfJoin, AdjacencyList, InvertedIndex, SequenceCount benchmarks from Tarazu benchmark suite [12]. Tarazu benchmarks represent typical jobs in production clusters. Since Hadoop-A specifically aims to improve I/O during the intermediate data shuffling, among different benchmarks, we focus on the data-intensive Terasort, which is a *de facto* standard Hadoop I/O benchmark, in which the sizes of intermediate data and final output are as large as the input size. TeraSort generates a lot of intermediate data and can expose the I/O bottleneck across the Hadoop data processing pipeline. WordCount counts the occurrence of words in the input data and generates relatively smaller intermediate data.

Many names are used in the experiments to describe the test cases, for example, *Hadoop on SDP* means we run original Hadoop on InfiniBand through Socket Direct Protocol. To avoid confusion, we list the protocol and network environment used for each test case in Table 5.1.

Table 5.1: Test Case Description

Test Cases	**Transport Protocol**	**Network**
Hadoop on 1GigE	TCP/IP	1GigE
Hadoop on 10GigE	TCP/IP	10GigE
Hadoop on IPoIB	IPoIB	InfiniBand
Hadoop on SDP	SDP	InfiniBand
JBS on 10GigE	TCP/IP	10GigE
JBS on IPoIB	IPoIB	InfiniBand
JBS on RoCE	RoCE	10GigE
JBS on RDMA	RDMA	InfiniBand

5.2.1 Benefits of JVM-Bypass

We start our experiments by examining the efficiency of JBS for intermediate data of different sizes. We run Tersort jobs of different input sizes on both InfiniBand and Ethernet clusters. For each data size, we conduct 3 experiments and report the average job execution time.

Figure 5.1 shows the experimental results. In the InfiniBand environment, compared to Hadoop on IPoIB and Hadoop on SDP, JBS on IPoIB reduces job execution time by 14.1% and 14.8%, respectively, on average. During these experiments, we notice that the performance of Hadoop on IPoIB is very close to that of Hadoop on SDP. Therefore, in the following sections, we use mostly Hadoop on IPoIB as the reference case to show the benefits of JBS in the InfiniBand environment.

In the Ethernet environment, JBS on 1GigE and JBS on 10GigE reduce the job execution times by 20.9% and 19.3% on average, compared to Hadoop on 1GigE and Hadoop on 10GigE, respectively.

In both environments, we observe that Hadoop on high-speed networks can speed up jobs with small sizes of intermediate data ($\leq$ 64GB). For instance, when the data size is 32GB, compared to Hadoop on 1GigE, Hadoop on IPoIB and Hadoop on 10GigE achieve improvements of 55.2% and 51.5%, respectively, on average. This is because the movement of small size data is less dependent on disks and most of them reside in disk cache or system buffers. Thus high-performance networks can exhibit better benefits for data shuffling.

Nevertheless, even with high-speed networks, Hadoop is still constrained from exploiting the full performance potentials of network hardware by the presence of JVM. In contrast, JBS eliminates the overhead of JVM, and improves the jobs with small size intermediate data. For 32GB input, JBS reduces the execution time by 11.1% and 12.7% on average, compared to Hadoop on IPoIB and Hadoop on 10GigE, respectively. For jobs with even smaller data sets, the costs of task initialization and destruction become dominant, JBS does not exhibit benefits.

Furthermore, as shown in Figure 5.1, the cases using high-performance networks (Hadoop on IPoIB and Hadoop on 10GigE) without JBS, compared to the Hadoop on 1GigE. do not provide noticeable improvements for large data sets ($\geq$ 128GB) due to disk I/O bottleneck caused by large data sets. JBS alleviates these issues through its batched and pipelined prefetching. Therefore, with a data size of 256GB, JBS on IPoIB and JBS on 10GigE improve the performance by 21.7% and 26.5%, respectively on average, compared to Hadoop on IPoIB and Hadoop and 10GigE.

Another interesting observation as shown in Figure 5.1(b) is that when data size grows close to 256GB, JBS performs similarly on 1GigE and 10GigE. This is because that the overhead incurred by large amount of memory copies for TCP/IP transportation becomes a severe bottleneck. Therefore, in the next section, we explore the benefit provided by RDMA protocol.

5.2.2 Benefits of RDMA

To evaluate the benefit of RDMA, we compare the performance of JBS on RDMA and JBS on RoCE to that of JBS on IPoIB and JBS on 10GigE. Figure 5.6 shows the experiment results. In the InfiniBand environment, JBS on RDMA outperforms JBS on IPoIB by 25.8% on average. In the Ethernet environment, compared to JBS on 10GigE, JBS on RoCE speeds up the job executions by 15.3% on average. Note that in both environments, running JBS on RDMA and RoCE achieve better performance for all data sizes.

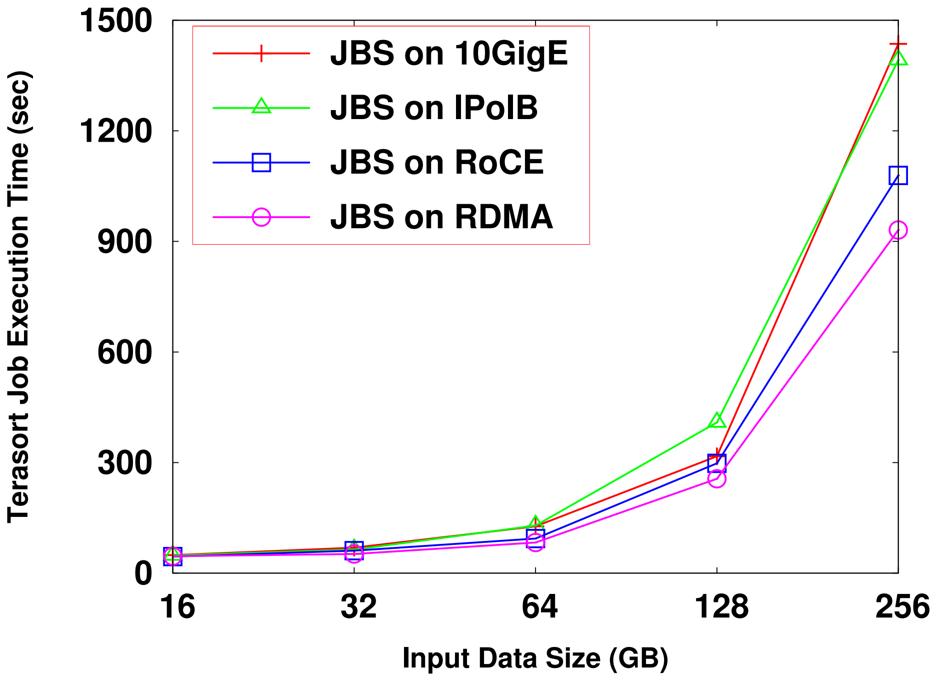
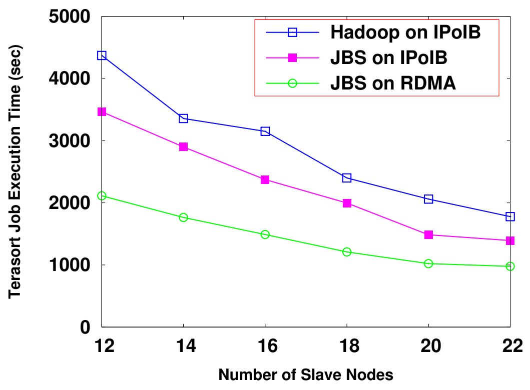
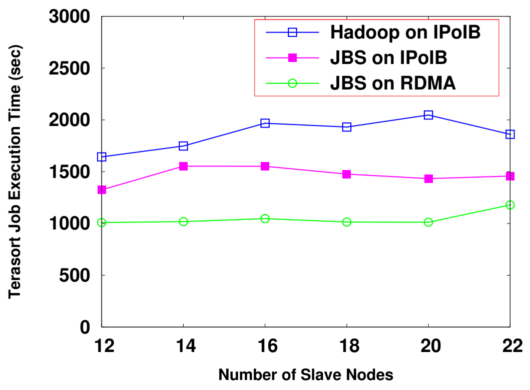


Figure 5.6: Benefits of RDMA

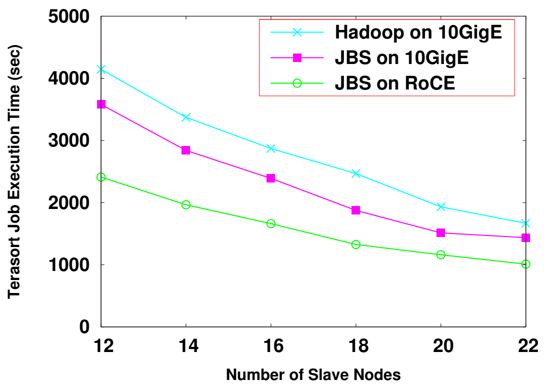
The reason that JBS delivers better performance with RDMA protocol is two-fold. Firstly, RDMA has significant performance advantages over TCP/IP because of its higher bandwidth and lower latency. Secondly, RDMA also reduces the number of memory copies for the movement of intermediate data.



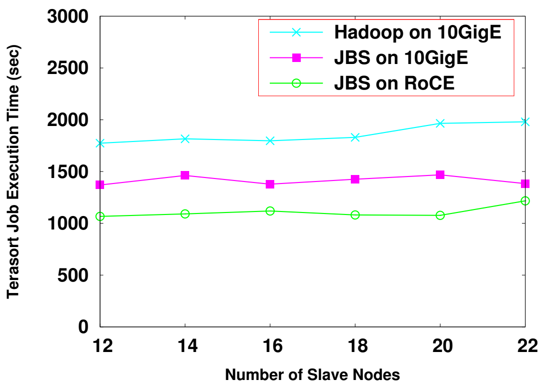
(a) Strong Scaling in the InfiniBand Environment



(b) Weak Scaling in the InfiniBand Environment



(c) Strong Scaling in the Ethernet Environment



(d) Weak Scaling in the Ethernet Environment

Figure 5.7: Scalability Evaluation

5.2.3 Scalability

High scalability is a critical feature that leads Hadoop to its success. JBS aims to preserve this feature. In this section, we evaluate the scalability of JBS by two scaling patterns: *Strong Scaling* and *Weak Scaling*. In the case of strong scaling, we use a fixed-size data (256GB) as Terasort input while increasing the number of compute nodes. In the case of weak scaling, we use a fixed-size data (6GB) for each ReduceTask of a Terasort job, so the total input size increases linearly when we increase the number of nodes, reaching 264GB when 22 slave nodes are used.

Figure 5.7(a) shows the results of the strong scaling experiments on the InfiniBand cluster. On average, JBS on RDMA and JBS on IPoIB outperform Hadoop on IPoIB by 49.5% and 20.9%, respectively. It achieves a linear reduction of the execution time with an increasing number of slave nodes. Figure 5.7(b) shows the results of the weak scaling experiments. JBS on RDMA and JBS on IPoIB reduce the execution time by 43.6% and 21.1%, respectively on average, compared to Hadoop on IPoIB. As also shown in the Figure, JBS maintains stable improvement ratios with a varying number of slave nodes.

Figures 5.7(c) and (d) show the results of scaling experiments in the Ethernet environment. JBS accomplishes similar performance improvement as in the InfiniBand environment. Compared to Hadoop on 10GigE, JBS on RoCE reduces the execution time by up to 41.9% for strong scaling tests and up to 40.4% for weak scaling tests on average. Compared to Hadoop on 10GigE, JBS on 10GigE reduces the execution time by 17.6% and 23.8%, respectively on average, for strong and weak scaling tests. Taken together, our results demonstrates that JBS is capable of providing better scalability than Hadoop in terms of the job execution time regardless of the underlying networks and protocols.

5.2.4 Differences in Terms of Network Communication

In this subsection, we have compared the time used to create and destroy network connection under RDMA and TCP/IP protocol. The feature of RDMA protocol is measured over Infiniband and Converged Ethernet (RoCE) respectively. As shown in figure 5.8, the costs for setting up and tearing down RDMA connection are much higher than the network connection using TCP/IP protocol. Under Infiniband environment, Infiniband IPoIB is able to achieve 95.4% and 90.4% better performance than Infiniband RDMA protocol on creating and destroying network connection respectively. On gigabit ethernet, RDMA and TCP/IP protocol performs similarly as on Infiniband.

Since the cost of setting up and tearing down RDMA network connection is high, as described in section 4.2.2 we keep RDMA connection when a new connection are created and reuse the connection if necessary. On the other side, due to the low cost of, a TCP/IP network connection are created or destroyed on demand.

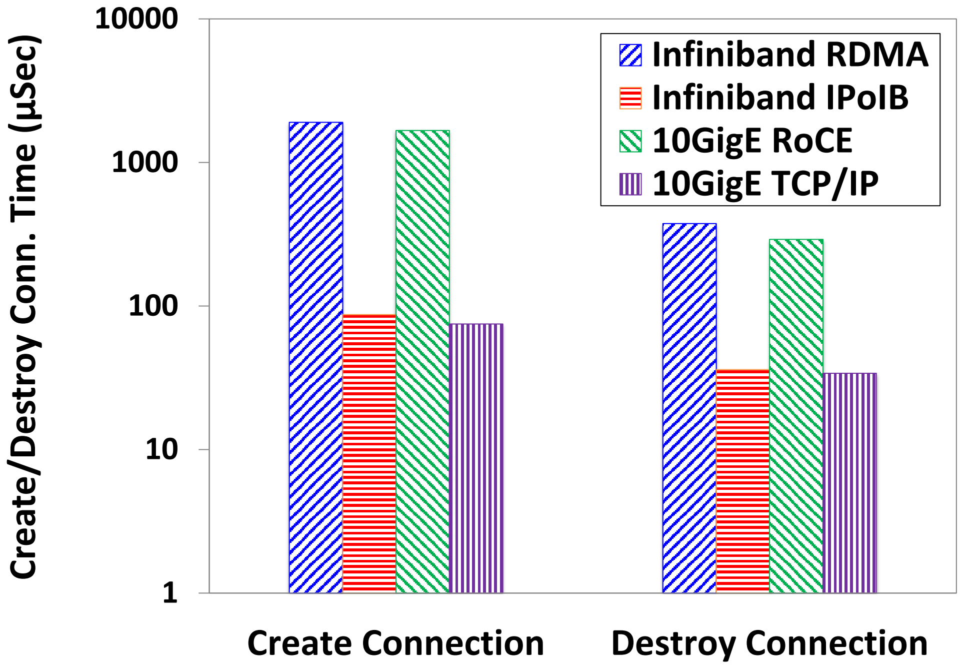
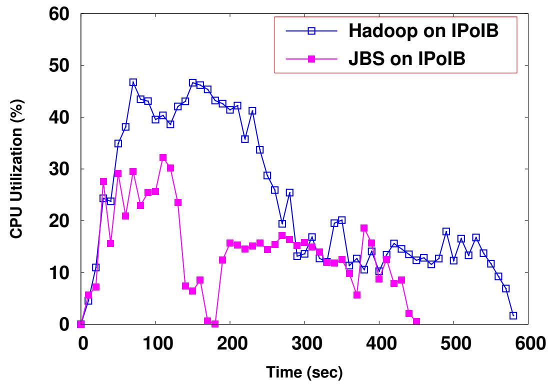


Figure 5.8: Comparation of Different Network Connection

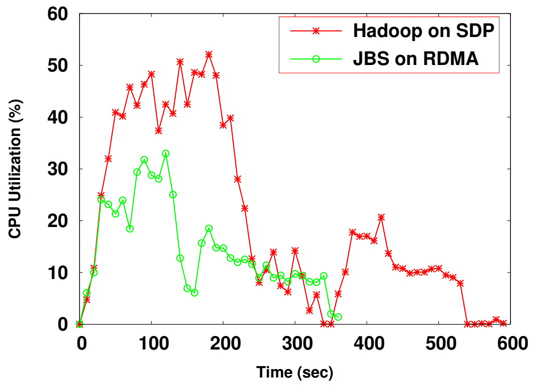
5.2.5 CPU Utilization

CPU utilization is another important performance metric. In this section, we measure the CPU utilization of JBS by using Terasort benchmark with 128GB input data. Low CPU utilization during data shuffling can spare more CPU cycles for Hadoop applications. On each node, we run *sar* in the background to collect CPU statistics and trace the output every 5 seconds. In the results, we report the average CPU utilization across all 22 slave nodes.

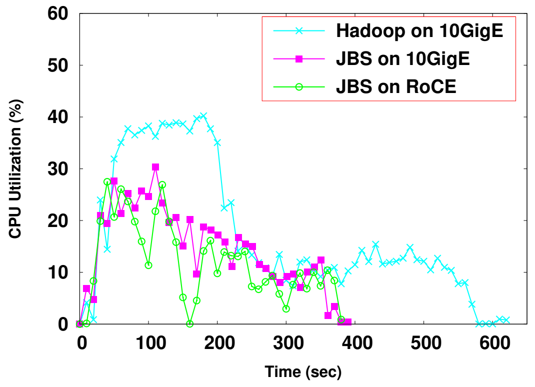
Figures 5.9(a) and (b) show the results of comparing CPU utilization between Hadoop and JBS in the InfiniBand environment. For fair comparison, we only consider CPU utilization in the same execution period. By eliminating the overhead of JVM and reducing the disk I/O, JBS on IPoIB greatly lowers the CPU utilization by 48.1% compared to Hadoop on IPoIB. In addition, even though the SDP protocol provides Java stream sockets to take advantage of RDMA through the socket interface, Hadoop on SDP can only reduce CPU utilization by 15.8%, compared to Hadoop on IPoIB. In contrast, JBS on RDMA significantly reduces the CPU utilization by 44.8%, compared to Hadoop on SDP. Note that, besides the RDMA benefit of low CPU utilization, another major factor that contributes to low utilization is the use of less number of threads in JBS. Unlike the original Hadoop in which each ReduceTask spawns more than 8 JVM threads for the purpose of data shuffling, JBS only requires 3 native C threads for the same.



(a) InfiniBand Environment (TCP/IP Protocol)



(b) InfiniBand Environment (RDMA Protocol)



(c) Ethernet Environment

Figure 5.9: CPU Utilization

Figure 5.9(c) shows the CPU utilization of JBS in the Ethernet environment. Compared to Hadoop on 10GigE, JBS on RoCE and JBS on 10GigE reduce the CPU utilization by 46.4% and 33.9%, respectively on average. In addition, JBS on RoCE reduces CPU utilization by about 18.7% due to RoCE's low CPU utilization and less memory copies.

Taken together, these results adequately demonstrate that JBS not only reduces the job execution time, but also achieves much lower CPU utilization.

5.2.6 Memory Utilization

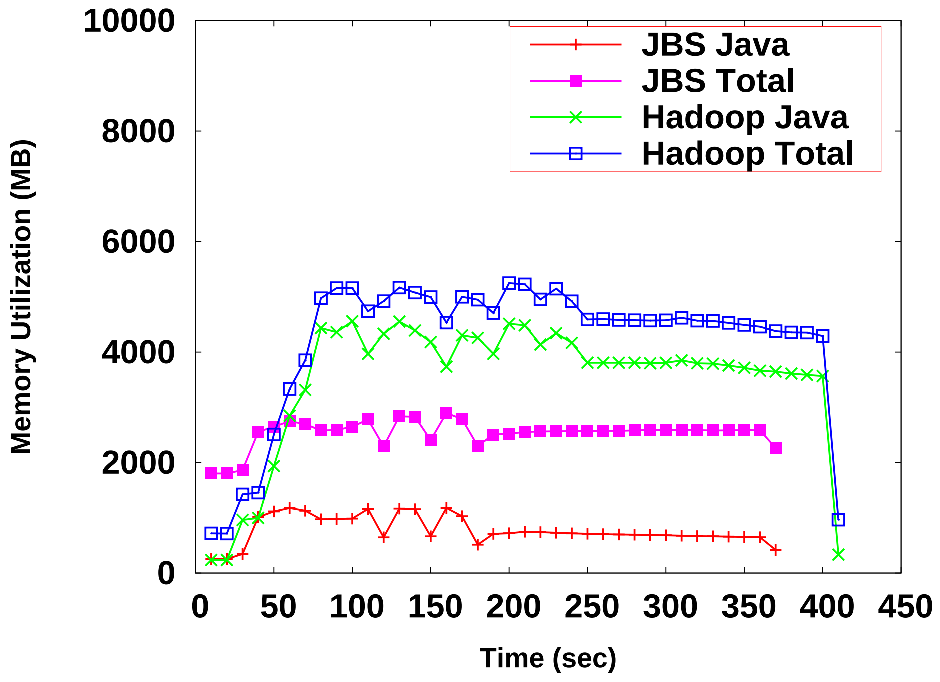


Figure 5.10: Memory Utilization

One of the most important objective of achieving JVM Bypass in Hadoop is reducing JVM memory usage overhead. In this section we have compared memory usage of JBS with Hadoop. The experiments are conducted on 60 GB input data by using Terasort benchmark on IPoIB, 10 slave nodes are utilized. On each node free and ps commands are leveraged to investigate total memory and Java memory usage respectively.

As can be seen in figure 5.10, JBS is able to reduce total memory usage by up to 39.7% on average than Hadoop. Specifically, further study shows that JBS's java program is using 77.8% less memory than original Hadoop. Such results indicate that through JVM bypass shuffling in Hadoop, JBS has mitigated JVM overhead significantly.

5.2.7 Impact of JBS Transport Buffer Size

The size of the buffer for network transportation has critical impact on the performance. Large buffer size can better utilize the bandwidth and reduce overheads due to less number of fetch requests for each segment, but it also results in less number of available buffers to be shared by data threads, causing more resource contention. To understand the impact of the buffer size, we measure the execution time of Terasort with 128GB input while changing the buffer size.

Figure 5.11 shows the results. For JBS on RDMA and JBS on RoCE, the execution time goes down with an increasing transport buffer size, and gradually levels off from 128KB and beyond. Compared to the 8KB buffer size, the 256KB buffer size improves the performance of JBS on RDMA by 53%. This performance difference is more significant for JBS on IPoIB. When the buffer size increases from 8KB to 128KB, execution time is reduced by up to 70.3%. However, when the size reaches 512KB, the performance is slightly degraded. This is because the use of very large buffers increases the contention between communication threads, and reduces the pipelining effects of many buffers.

Overall this evaluation demonstrates that a large buffer size up to 128KB can effectively improve job execution time. For this reason, we choose the default transport buffer size as 128KB for the JBS library.

5.2.8 Effectiveness on Different Benchmarks

To assess the optimization effectiveness of JBS to other Hadoop applications, we have evaluated JBS with Tarazu benchmark suite in addition to Terasort. The input for those benchmarks are 30GB, using either wikipedia data or database data. Figures 5.12 (a) and (b) present the job execution times in both InfiniBand and Ethernet environments. Overall, these benchmarks can be categorized into two types.

For the first type of benchmarks including SelfJoin, InvertedIndex, SequenceCount, and AdjacencyList, each MapTask generates a lot of intermediate data to be shuffled to ReduceTasks.

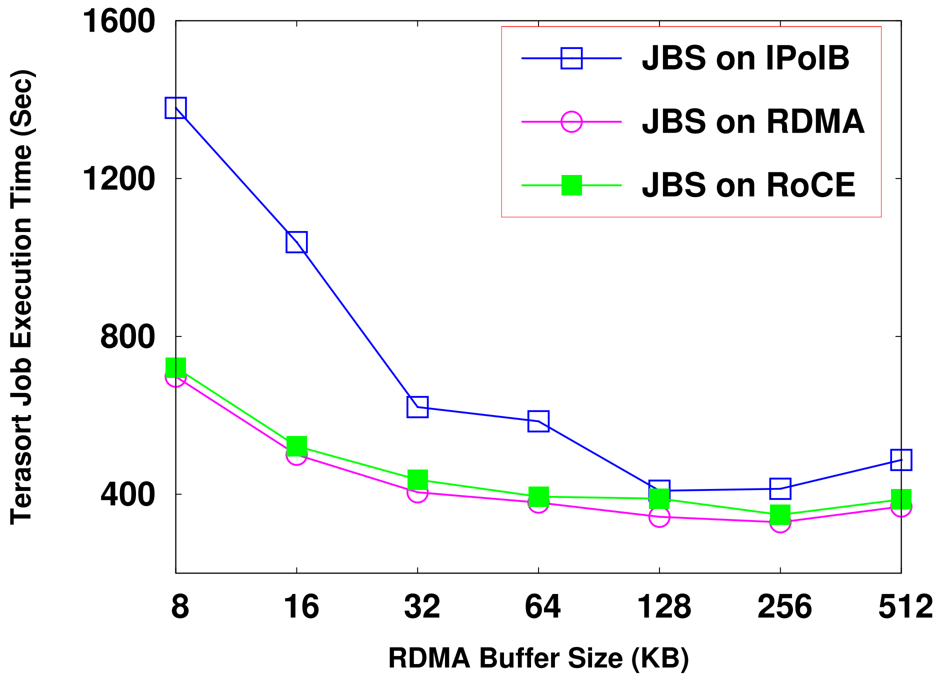
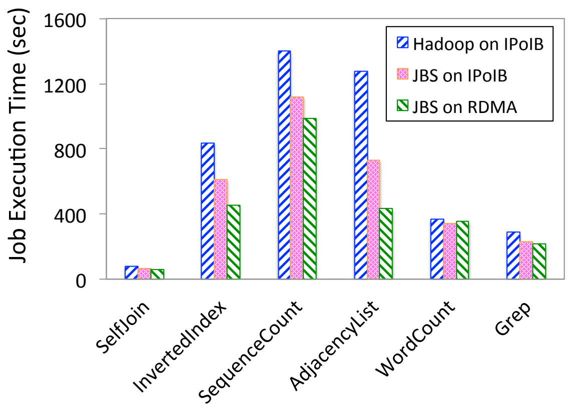


Figure 5.11: Impact of Different RDMA Buffer Sizes

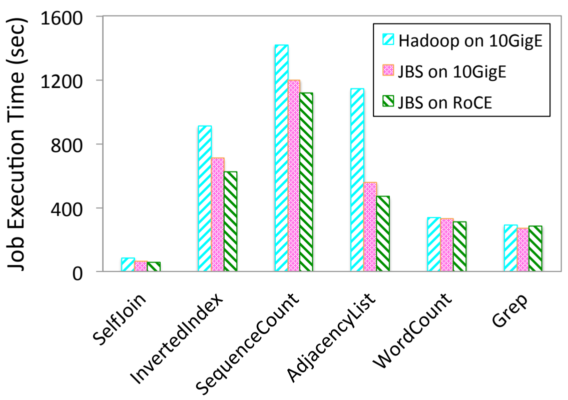
Because of its strength in accelerating the shuffling phase of Hadoop, JBS is geared to provide good performance benefits for these applications.

In the InfiniBand environment, we observe that JBS on RDMA achieves an average of 41% reduction in the execution time for these four benchmarks, and reaches up to 66.3% improvement for AdjacencyList. JBS on IPoIB reduces the execution times of these benchmarks by 26.9% on average. In the Ethernet environment, compared to Hadoop on 10GigE, JBS on RoCE reduces the execution times by 36.1% on average for these benchmarks. JBS on 10GigE only reduces the execution times by 29.8% on averaged compared to the same.

In contrast, for the second type of benchmarks, WordCount and Grep, only a small amount of intermediate data is generated. As a result, JBS does not gain performance improvement for these two benchmarks.



(a) InfiniBand Environment



(b) Ethernet Environment

Figure 5.12: Performance of Different Benchmarks

5.3 Effectiveness of VAS Framework

In this section, we evaluate the performance of our VAS framework compared to the original YARN. We first describe our experimental environment, then we provide our experimental results in detail.

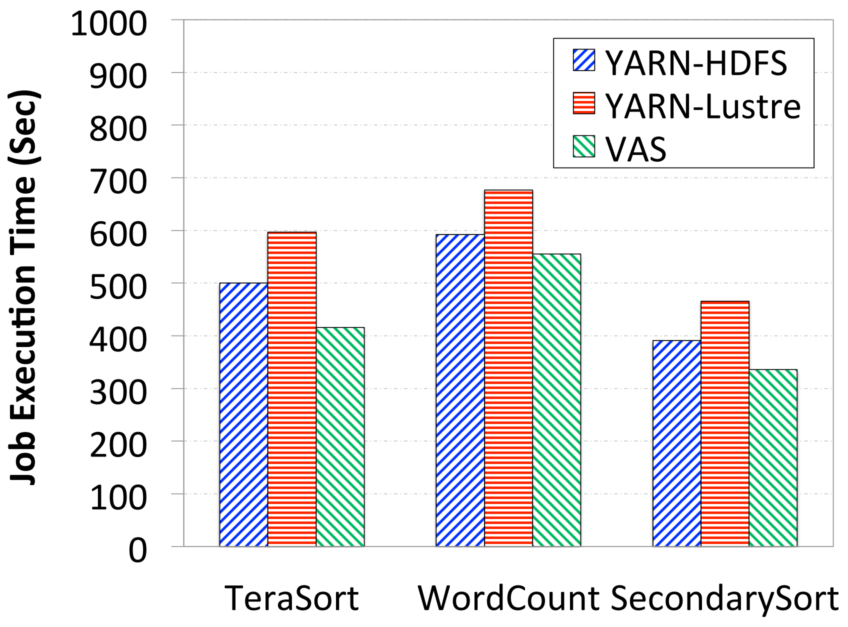
Our experiments are conducted in a 22-node cluster. Each node contains dual-socket quad-core 2.13 GHz Intel Xeon processors and 8GB DDR2 800 MHz memory, along with 8x PCI-Express Gen 2.0 bus. Each node has a 500GB, 7200 RPM Western Digital SATA hard-drivers. All nodes are equipped with Mellanox ConnectX-2 QDR Host Channel Adaptors, which are configured to run in the 10G mode and connect to a 36-port 10G switch. We install InfiniBand software stack and Mellanox OFED version 2.0, with SR-IOV enabled.

In our evaluation, all experiments are based on Hadoop 2.0.4 and Lustre 2.5.0. Each container is allocated 1GB memory and the maximum memory configured for the virtual machine on each node is 6GB. A group of benchmarks have been leveraged for performance measurements, including Terasort, WordCount, SecondarySort and TestDFSIO in the YARN distribution. Among them, Terasort and SecondarySort generate a large amount of intermediate data. WordCount produces only a small quantity of intermediate data. TestDFSIO benchmark is designed to test the raw performance of a file system, in which the write and read throughput are measured. These diverse benchmarks are used to evaluate our framework from a variety of aspects.

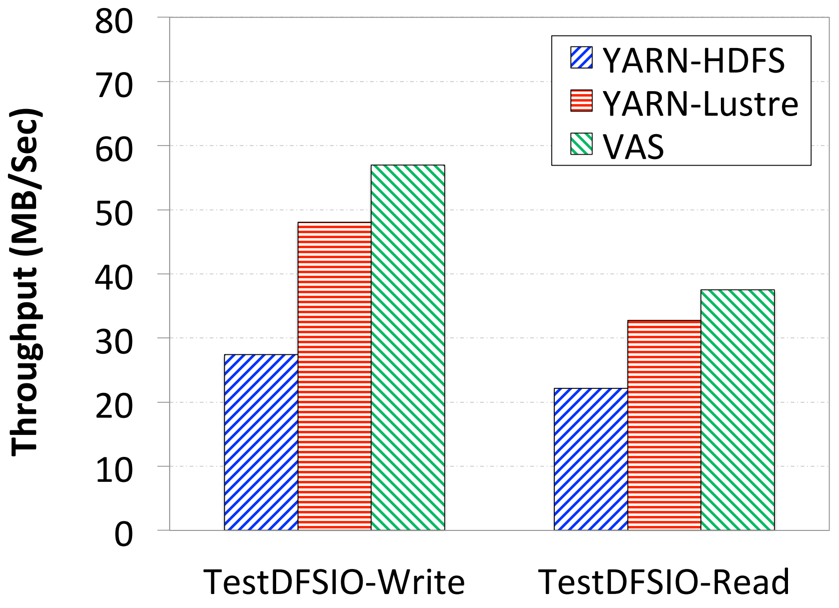
5.3.1 Overall Performance

Fig. 5.13 compares the overall performance of YARN on HDFS (YARN-HDFS), YARN on Lustre (YARN-Lustre) and our VAS framework (VAS) for the four benchmarks. Fig. 5.13(a) shows the job execution time of TeraSort, WordCount and SecondarySort. For all three benchmarks, YARN-Lustre performs the worst. That is because simply porting MapReduce analysis framework to work with Lustre cannot fully leverage the advanced features of Lustre and suffers from lock contention for the access of intermediate data. YARN-HDFS is 12.6% to 16.2% faster than YARN-Lustre because it leverages map input locality for fast data movement. On average, our VAS framework achieves an execution time 12.4% and 25.4% better than YARN-HDFS and YARN-Lustre, respectively. Specifically, VAS outperforms YARN-HDFS and YARN-Lustre by 16.8% and 30.3% for Terasort benchmark. Such performance gains demonstrate the strengths of our techniques in stripe-aligned data placement, task scheduling, and pipelined merging and reducing.

To shed light on how different schemes can leverage Lustre for I/O, we further measure the average I/O throughput using TestDFSIO. As shown in Fig. 5.13(b), VAS achieves 14.6% and 69.7% better read bandwidth and 18.7% and 108% better write bandwidth compared to YARN-Lustre and YARN-HDFS, respectively. In general, DFSIO displays better performance on Lustre than on HDFS. This is because of the I/O capability of Lustre. In addition, the performance improvement of VAS over YARN-Lustre comes from our exploitation of data locality and optimization on intermediate data shuffing. Since HPC uses parallel file systems such as Lustre instead of HDFS, in the rest of this paper, we focus on the comparison between VAS and YARN-Lustre, omitting the case of YARN-HDFS.



(a) Job Execution

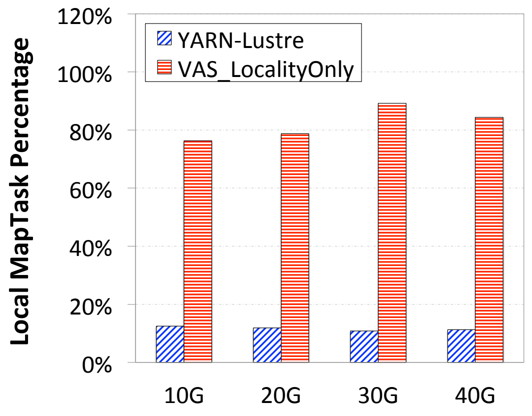


(b) TestDFSIO

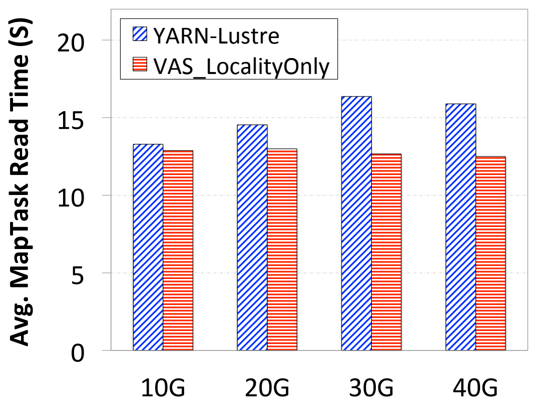
Figure 5.13: The Overall Performance of Three Schemes

5.3.2 Locality Improvement and the Benefit

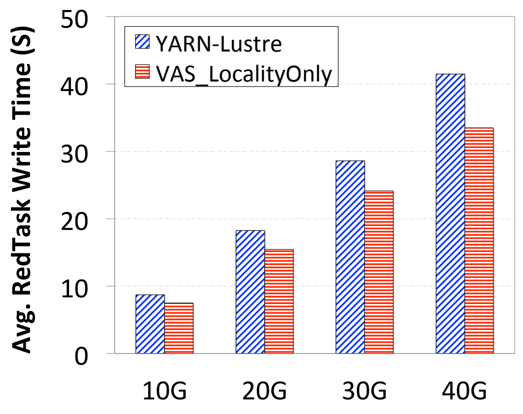
We have investigated the benefits of data locality in VAS. VAS allocates a MapTask to an OSS with the input split on its local OST. In addition, MapTasks will be controlled to dump its spill data



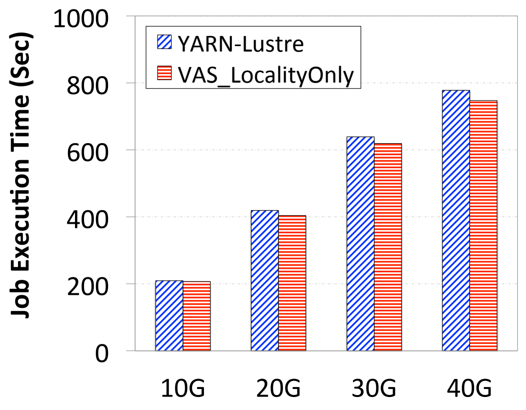
(a) Percentage of Local Tasks



(b) MapTask Read Time



(c) ReduceTask Write Time



(d) Job Execution Time

Figure 5.14: Locality Improvement

and MOFs to local OSTs. Similarly, ReduceTasks write any of its spill/merge data and output to local OSTs. Here we focus on the comparison between YARN-Lustre and our VAS framework that is enabled with only the locality optimization (VAS-Locality-Only), i.e., without our technique on pipelined merging as detailed in Section 4.3.3. We run TeraSort on 10 slave nodes with input data varying from 10GB to 40GB. During the tests, we measure the percentage of local MapTasks, and profile the performance of I/O operations and job execution time. Fig. 5.14(a) shows the percentage of local MapTasks. As shown in the figure, YARN-Lustre randomly launches MapTasks, achieving only 11.6% in terms of the percentage of local MapTasks. In contrast, our VAS framework achieves high percentage of local MapTasks by effectively managing the distribution of data splits. VAS-Lustre schedules 82.1% of MapTasks with its data on the local OST.

To examine the performance benefits of data locality, we have measured the average input read time by MapTasks. As shown in Fig. 5.14(b), the performance gain is not significant for 10 GB data size. This is because the virtual VMs are connected with 10 Gigabit Ethernet. Data locality cannot bring much benefit when the data size is small. As the data size increases to 40GB, the MapTask reading time in VAS outperforms YARN by as much as 21.4%. On average, the reading performance of MapTasks in VAS is 15% better than YARN-Lustre due to improved data locality.

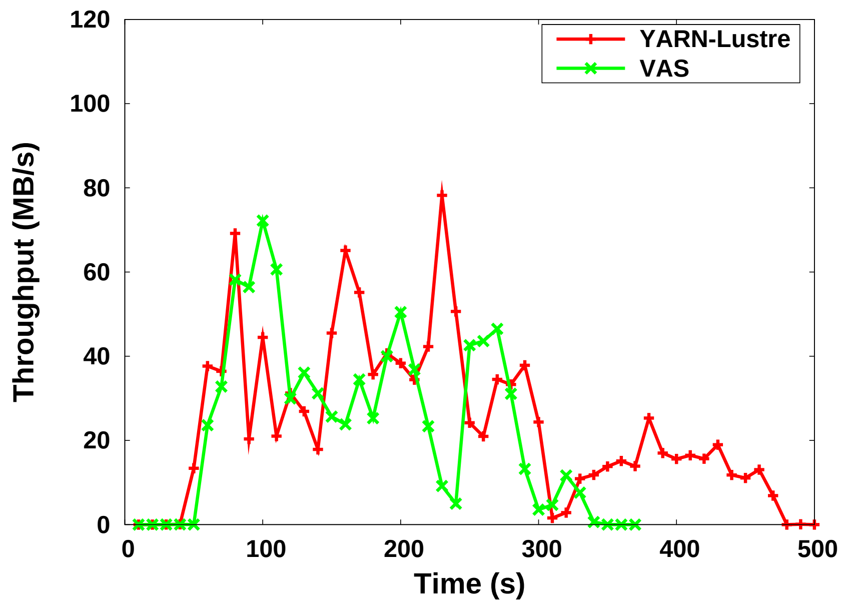
Fig. 5.14(c) presents the write performance of ReduceTasks. In YARN, since the number of ReduceTasks is fixed while the number of MapTasks is proportional to the size of input data. The writing time of ReduceTasks grows with an increasing size of total data. As shown in the figure, VAS-Locality-Only leads to 19.3% better write performance than YARN-Lustre when the data size is 40GB. This performance improvement comes from the better locality in spilling data and storing output. Moreover, the result also demonstrates that leveraging data locality over Lustre can obtain better write performance for the ReduceTasks.

Fig. 5.14(d) shows the job execution time. VAS-Locality-Only demonstrates up to 4% better job execution time than YARN-Lustre for a varying amount of input data. The overall percentage of improvement on the job execution time is much smaller because the weights of the MapTask reading time and the ReduceTask writing time are quite low in the total job execution time.

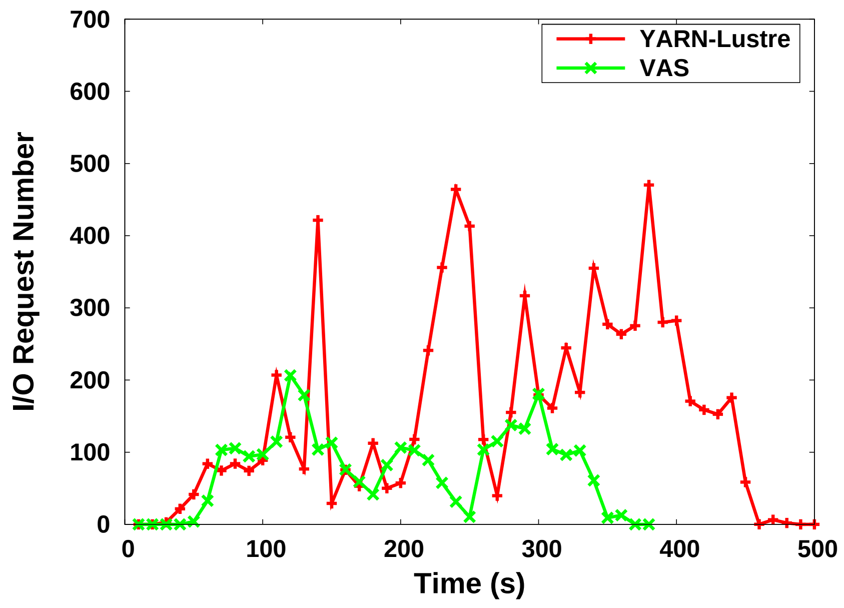
5.3.3 Network and Disk I/O Utilization

We also examine the effectiveness of network and disk I/O utilization. These experiments are conducted with TeraSort running on 30GB dataset. Fig. 5.15 shows the network throughput and I/O request numbers during the MapReduce job execution. In Fig. 5.15(a), VAS cuts down the total network throughput by up to 26.7% since it introduces the stripe-aligned I/O and task scheduling.

Fig.5.15(b) shows the I/O request numbers during execution. VAS significantly reduces the disk I/O request numbers by 62.3% on average compared to YARN-Lustre. This is because the avoidance of explicit shuffling in VAS, which eliminates the associated data spilling and retrieval. Therefore, at the beginning of job execution (namely the map phase), I/O requests in the two



(a) Network Throughput



(b) Number of I/O Requests

Figure 5.15: Profile of Resource Utilization

cases are very close to each other. When the job further progresses to the reduce phase, we can observe that the I/O request number of VAS becomes much lower than YARN-Lustre. These results indicate that our techniques not only improves the shuffle phases but also provides an efficient pipeline of data merging and reducing, which can achieve better job execution.

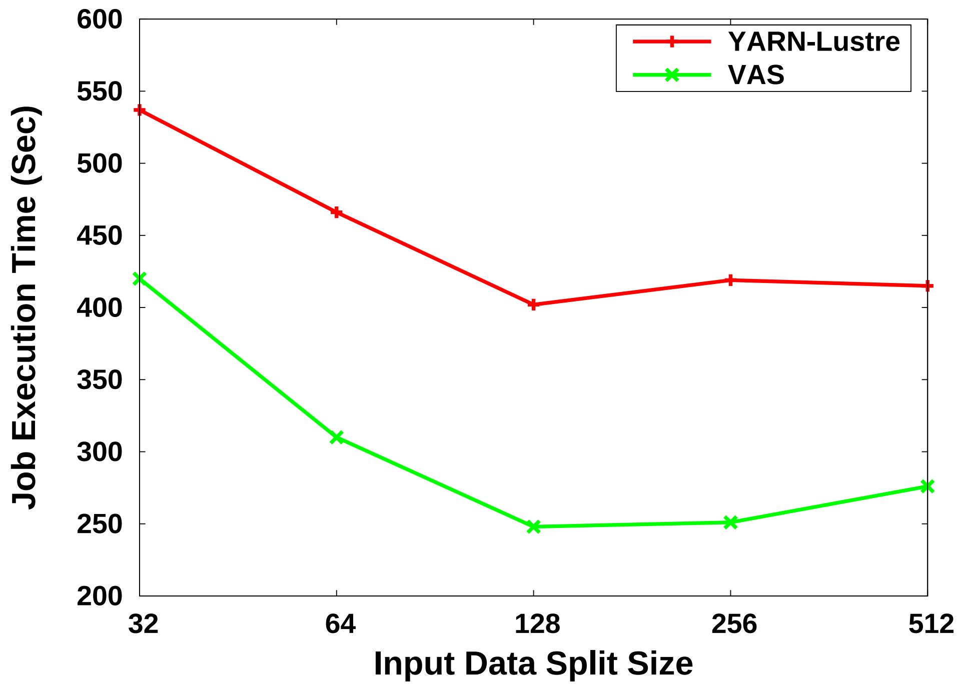


Figure 5.16: Tuning of the Input Split Size

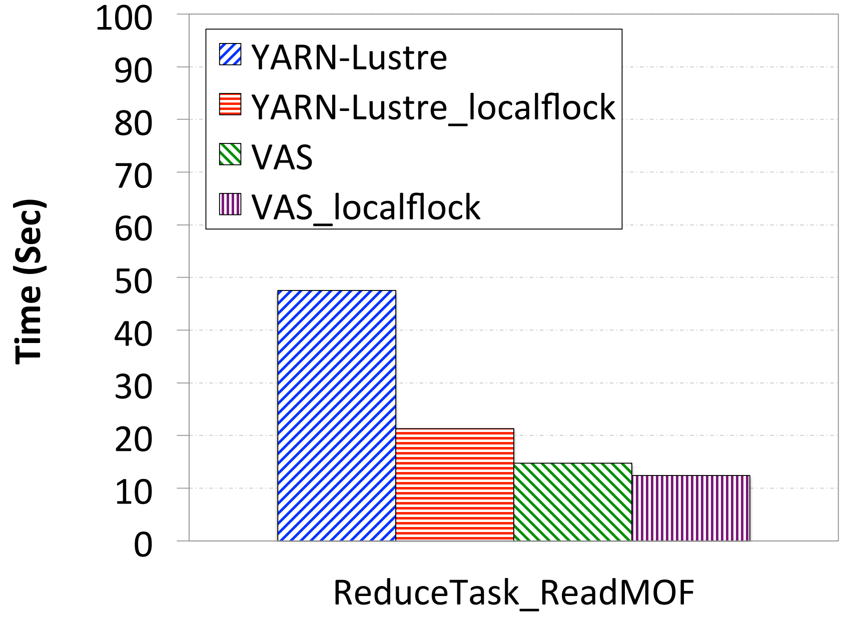
5.3.4 Tuning of Input Split Size

Since we have determined to set Lustre stripe size equal to map input split size, we aim to gain more insights on how the map input split size affects the execution performance of MapReduce jobs. In our VAS framework, MapTasks in YARN take input data from Lustre. The choice of the input split size represents an important tradeoff between data access sequentiality and task execution parallelism. To be specific, a split size too small can increase scheduling overheads, while a split size too big may reduce the concurrency of task execution. Fig. 5.16 shows the execution time of TeraSort for YARN-Lustre and VAS with the input split size varying from 32 MB to 512 MB. As we can see, both schemes perform best when input split size is set as 128 MB. These results suggest that 128 MB provides the best tradeoff between task parallelism, management overhead and I/O granularity.

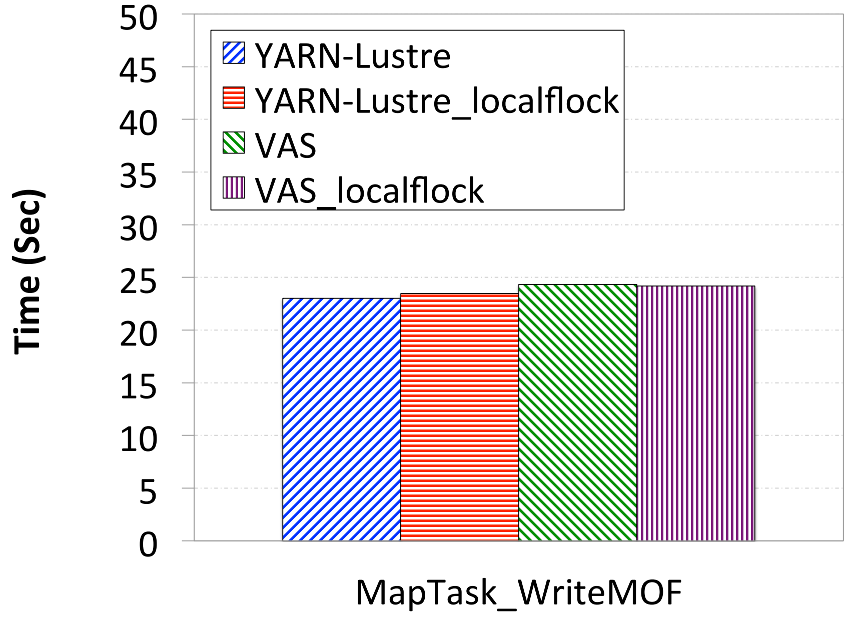
5.3.5 The Impact of Lock Management

Lustre distributed lock management is known as a critical factor for the parallel I/O performance. In our VAS framework, many I/O operations from many map and reduce tasks overlap with each other, thereby presenting a scenario of parallel I/O. To alleviate the impact of Lustre lock contention on the central MDS, there is an option (*localflock*) to delegate the management of Lustre locks to each OSS.

We have also measured the I/O performance of four different execution cases: YARN-Lustre, YARN-Lustre with localflock enabled (YARN-Lustre-localflock), VAS, and VAS with localflock enabled (VAS-localflock). Fig. 5.17 shows the MOF reading time for ReduceTasks and the MOF writing time for MapTasks when running TeraSort on 30GB data. Fig. 5.17(a) shows that VAS with the localflock option (VAS-localflock) can improve the performance of reading intermediate files in ReduceTasks by up to 55.2% and 15.9%, compared to YARN-Lustre-localflock and VAS, respectively. To be specific, YARN-Lustre-localflock reduces the read time of MOFs by 69% compared to YARN-Lustre. This performance difference is due to the serious lock contention when many ReduceTasks are reading the same MOFs. The localflock option can greatly alleviate the impact of such contention. However, the option of localflock does not have a significant performance impact for the VAS framework. This can be explained by two reasons. First, our pipelined merging/reducing technique reduces the frequency of I/O operations. Second, VAS has modified the original MOF format. In the original YARN MapReduce, each MapTask generates one MOF file while ReduceTasks fetch their segments according to the offsets in the MOF file. In the VAS framework, each MapTask creates separate files, one for each ReduceTask. This avoids the lock contention issue on a shared MOF file. As depicted in Fig. 5.17(b), VAS-localflock slightly increases the average write time. This is because VAS leads to more metadata operations because of the creation of more intermediate files.



(a) MOF Reading Time in ReduceTasks



(b) MOF Writing Time in MapTasks

Figure 5.17: Impact of Lustre Lock Management

Chapter 6
Summary

In this dissertation, we spend our efforts on optimizing communication and data movement on HPC and Cloud Systems, including speeding up MPI AlltoallV collective operation, optimizing Hadoop data shuffling and achieving efficient analytics shipping.

Scalable LOgarithmic AlltoallV Algorithm for Hierarchical Multicore Systems: The linear complexity in existing MPI AlltoallV operations severely hinders the scalability of scientific applications. In this study, we introduce a *S*calable *LO*garithmic *A*lltoall*V* algorithm, named as *SLOAV*, to reduce the complexity of AlltoallV collective communication for small messages of different sizes to logarithmic complexity. With such algorithm, the number of necessary messages for AlltoallV can be significantly reduced, thus the scalability of scientific applications can be effectively improved. Furthermore, to leverage the advantages of shared memory on multicore systems, we design the *SLOAVx* algorithm based on SLOAV. SLOAVx elects a group leader for all the processes on the same node and delegates the leaders on all the nodes to conduct inter-node communication via SLOAV. Both SLOAV and SLOAVx have been implemented and integrated into Open MPI. To assess their efficiency, we have systematically evaluated their performance on the Jaguar supercomputer and the Smoky cluster at Oak Ridge National Lab. Our experimental results demonstrate that SLOAV can significantly reduce the latency by up to 86.4%, when compared to the existing implementations. SLOAVx can further improve the performance of SLOAV on multicore systems by as much as 83.1%.

JVM-Bypass for Efficient Hadoop Shuffling: In this dissertation, we have comprehensively analyzed the performance of Hadoop running on InfiniBand and 1/10 Gigabit Ethernet. Our experiment results reveal that simply switching to the high-performance interconnects cannot effectively boost the performance of Hadoop. To investigate the cause, We identify the overhead imposed by

JVM on Hadoop intermediate data shuffling. We have designed and implemented JVM-Bypass Shuffling (JBS) to avoid JVM in the critical path of Hadoop data shuffling. Our implementation of JBS also enables it as a portable library that can leverage both conventional TCP/IP protocol and high-performance RDMA protocol in different network environments. Our experimental evaluation demonstrates that JBS can effectively reduce the execution time of Hadoop jobs by up to 66.3% and lower the CPU utilization by 48.1%. Furthermore, with a set of different application benchmarks, we demonstrate that JBS can significantly reduce the CPU utilization and job execution time for Hadoop jobs that generate a large amount of intermediate data.

Exploiting Analytics Shipping with Virtualized MapReduce on HPC Backend Storage Servers: Simulation codes on large-scale HPC systems are generating gigantic datasets that need to be analyzed for scientific discovery. While in-situ technologies are developed for extraction of data statistics on the fly, there is still an open question on how to conduct efficient analytics of permanent datasets that have been stored to HPC backend storage. In this dissertation, we have exploited the analytics shipping model for fast analysis of large-scale persistent scientific datasets on backend storage servers of HPC systems without moving data back to the compute nodes. We have undertaken an effort to systematically examine a series of challenges for effective analytics shipping. Based on our examination, we have developed a Virtualized Analytics Shipping framework (VAS) as a conceptual prototype, using MapReduce as a representative analytics model and Lustre as a representative HPC backend storage system With the VAS framework, we have also provided several tuning and optimizations including fast network and disk I/O through dynamic routing, stripe-aligned data distribution and task scheduling, and pipelined intermediate data merging and reducing. Together, these techniques have realized an efficient analytics shipping implementation, which supports fast and virtualized MapReduce programs on backend Lustre storage servers.

Chapter 7

Future Plan

In this dissertation, SLOAVx, JBS and the VAS framework are proposed to address the communication and data movement issues in the existing MPI library and MapReduce framework. There are several avenues for future research.

In terms of MPI collective communication, I plan to extend my research in respect to MPI AlltoallV operation to improve the efficiency of exchanging large messages. The time of transmitting message consists of start-up cost and data transfer time. Start-up cost is the most important factor impacting the performance of small message transmission. Contrarily, data transfer time is the dominant cost of transmitting large messages. Thus the large messages operation does not fit the logarithmic algorithm, in which the same message is transferred many times. To this end, I plan to exploit other mechanisms such as reducing network contention and balancing the load, to boost the performance of AlltoallV large message data transmission.

For the JVM-Bypass Shuffling study, Yarn is the next-generation of the Hadoop MapReduce framework, which is gaining rising focus in the area of big data analytics. Thus it is well worth porting our JBS efforts into the Yarn program. However, many challenges need to be addressed to achieve code porting. For instance, the architecture of data shuffling in Yarn has changed. TaskTracker and ReduceTask are responsible for end-to-end data transmission in Hadoop implementation. On the contrary, Yarn employs independent components to handle data shuffling on each node. For the above-mentioned reason, JVM-Bypass Shuffling needs to be modified to interact with the Yarn program and serve as a complementary element for data fetching.

In future work, further investigation will be conducted to measure the performance of the VAS framework. I plan to run HPC applications on Lustre clients, and at the same time launch MapReduce analytic applications in YARN. Two cases will be evaluated. In the first case, the

MapReduce programs are shipped to Lustre Storage Servers. This configuration represents the scenario in which a MapReduce program is shipped to the backend storage servers. In the second case (VAS), an additional cluster of Lustre Clients is configured to run MapReduce programs. This configuration represents the scenario in which a MapReduce program runs at the computing partition and has to retrieve data from HPC backend storage servers.

Bibliography

[1] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

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

[3] Chroot. https://wiki.archlinux.org/index.php/Change_Root.

[4] Gordon: Data-Intensive Supercomputing. http://www.sdsc.edu/supercomputing/gordon/.

[5] Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[6] Lustre Filesystem. http://www.lustre.org/.

[7] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[8] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/.

[9] Plugin for Generic Shuffle Service. https://issues.apache.org/jira/browse/MAPREDUCE-4049.

[10] The Xen Project. http://wiki.xen.org/wiki/Main_Page.

[11] VMware, Inc. http://www.vmware.com/.

[12] 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 '12, pages 61–74, New York, NY, USA, 2012. ACM.

[13] 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 1st USENIX Workshop on Hot Topics in Cloud Computing (HOTCLOUD'2009), San Diego, CA, USA*, 2009.

[14] Baidu, Inc. Hadoop C++ Enhancement. http://issues.apache.org/jira/ browse/MAPREDUCE-1270.

[15] J.C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, Tong Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, Hongfeng Yu, Fan Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9, Nov 2012.

[16] Brice Goglin and Stephanie Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing (JPDC)*, 2012.

[17] Ron Brightwell and Keith D. Underwood. An analysis of the impact of mpi overlap and independent progress. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 298–305, New York, NY, USA, 2004. ACM.

[18] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, SPAA '94, pages 298–309, New York, NY, USA, 1994. ACM.

[19] Castep Developers Group (CDG). Calculating the properties of materials from first principles. http://www.castep.org/, June 2012.

[20] 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.

[21] 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.

[22] 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.

[23] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More convenient more overhead: the performance evaluation of hadoop streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 307–313, New York, NY, USA, 2011. ACM.

[24] Ahmad Faraj and Xin Yuan. Communication characteristics in the nas parallel benchmarks. In Selim G. Akl and Teofilo F. Gonzalez, editors, *International Conference on Parallel and Distributed Computing Systems, PDCS 2002, November 4-6, 2002, Cambridge, USA*, pages 724–729. IASTED/ACTA Press, 2002.

[25] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 393–402, New York, NY, USA, 2005. ACM.

[26] Philip Werner Frey and Gustavo Alonso. Minimizing the hidden cost of rdma. In *ICDCS*, pages 553–560. IEEE Computer Society, 2009.

[27] Richard Graham, Manjunath Gorentla Venkata, Joshua Ladd, Pavel Shamis, Ishai Rabinovitz, Vasily Filipov, and Gilad Shainer. Cheetah: A framework for scalable hierarchical collective operations. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 73–83, Washington, DC, USA, 2011. IEEE Computer Society.

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

[29] 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.

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

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

[32] Nusrat S. Islam, Md W. Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'12. ACM, 2012.

[33] Adrian Jackson and Stephen Booth. Planned alltoallv. Technical report, EPCC (Edinburgh Parallel Computing Centre), July 2004.

[34] Robert J.Chanslet. Data availability and durability with the hadoop distributed file system. ;login' 12. USENIX Association, 2012.

[35] 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.

[36] 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.

[37] 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 the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 42. ACM, 2013.

[38] 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.

[39] Zhuo Liu, Bin Wang, Teng Wang, Yuan Tian, Cong Xu, Yandong Wang, Weikuan Yu, Carlos A Cruz, Shujia Zhou, Tom Clune, et al. Profiling and improving i/o performance of a large-scale climate scientific application. In *Computer Communications and Networks (ICCN), 2013 22nd International Conference on*, pages 1–7. IEEE, 2013.

[40] David Luan, Simon Huang, and GaoSheng Gong. Using Lustre with Apache Hadoop. *Sun Microsystems Inc.*, 2009.

[41] Teng Ma, George Bosilca, Aurelien Bouteiller, Brice Goglin, Jeffrey M. Squyres, and Jack J. Dongarra. Kernel assisted collective intra-node mpi communication among multi-core and many-core cpus. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 532–541, Washington, DC, USA, 2011. IEEE Computer Society.

[42] Carlos Maltzahn, Esteban Molina-Estolano, Amandeep Khurana, Alex J Nelson, Scott A Brandt, and Sage Weil. Ceph as a scalable alternative to the hadoop distributed file system. *login: The USENIX Magazine*, 2010.

[43] Mitchell Nick and Sevitsky Gary. Building memory-efficient java application:practices and challenges. PLDI '09. ACM, 2009.

[44] Ohio State University. Osu micro-benchmarks 3.7. http://mvapich.cse.ohio-state.edu/benchmarks/, September 2012.

[45] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 28:1–28:10, New York, NY, USA, 2007. ACM.

[46] Martin Plummer and Keith Refson. An lpar-customized mpi_alltoallv for the materials science code castep. Technical report, EPCC (Edinburgh Parallel Computing Centre), July 2004.

[47] 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*, pages 1–6. ACM, 2012.

[48] Sriram Rao. I-files: Handling Intermediate Data In Parallel Dataflow Graphs (Sailfish).

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

[50] 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, August 2009.

[51] Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, www.ispass.org, 28-30 March 2010, White Plains, NY, USA*, pages 122–133. IEEE Computer Society, 2010.

[52] Hongzhang Shan and John Shalf. Using IOR to analyze the I/O performance for HPC platforms. In Cray Users Group Meeting (CUG) 2007, Seattle, Washington, USA, 2007.

[53] 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.

[54] S. Sur, H. Wang, J. Huang, 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.

[55] Wittawat Tantisiriroj, S Patil, and G Gibson. The crossing the chasm: Sneaking a parallel file system into hadoop. In *SC08 Petascale Data Stroage Workshop*, 2008.

[56] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J Lang, Garth Gibson, and Robert B Ross. On the duality of data-intensive file system design: reconciling hdfs and pvfs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 67. ACM, 2011.

[57] Yuan Tian, Cong Xu, Weikuan Yu, Jeffrey S Vetter, Scott Klasky, Honggao Liu, and Saad Biaz. necodec: nearline data compression for scientific applications. *Cluster Computing*, 17(2):475–486, 2014.

[58] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 119–132, Berkeley, CA, USA, 2013. USENIX Association.

[59] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[60] Yandong Wang, Yizheng Jiao, Cong Xu, Xiaobing Li, Teng Wang, Xinyu Que, Cristi Cira, Bin Wang, Zhuo Liu, Bliss Bailey, et al. Assessing the performance impact of high-speed interconnects on mapreduce. In *Specifying Big Data Benchmarks*, pages 148–163. Springer, 2014.

[61] 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.

[62] Yandong Wang, Cong Xu, Xiaobing Li, and Weikuan Yu. Jvm-bypass for efficient hadoop shuffling. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 569–578. IEEE, 2013.

[63] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[64] Cong Xu. Tcp/ip implementation of hadoop acceleration, 2012.

[65] Cong Xu, RJ Goldstone, Zhuo Liu, Hui Chen, Bryon Neitzel, and Weikuan Yu. Exploiting analytics shipping with virtualized mapreduce on hpc backend storage servers. *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[66] 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 *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 369–376. IEEE, 2013.

[67] Weikuan Yu, Qi Gao, and Dhabaleswar K. Panda. Adaptive connection management for scalable mpi over infiniband. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 102–102, Washington, DC, USA, 2006. IEEE Computer Society.

[68] Weikuan Yu, D. K. Panda, and D. Buntinas. Scalable, high-performance nic-based all-to-all broadcast over myrinet/gm. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 125–134, Washington, DC, USA, 2004. IEEE Computer Society.

[69] Weikuan Yu, Yandong Wang, Xinyu Que, and Cong Xu. Virtual shuffling for efficient data movement in mapreduce. *IEEE Transactions on Computers*, 2013.

[70] Weikuan Yu, K John Wu, Wei-Shinn Ku, Cong Xu, and Juan Gao. Bmf: Bitmapped mass fingerprinting for fast protein identification. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 17–25. IEEE, 2011.

[71] 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.

[72] Fan Zhang, Solomon Lasluisa, Tong Jin, Ivan Rodero, Hoang Bui, and Manish Parashar. In-situ feature-based objects tracking for large-scale scientific simulations. In *SC Companion*, pages 736–740. IEEE Computer Society, 2012.