ATLANTIS PRESS
*Advances in Computer Science Research, volume 50*
4th International Conference on Electrical & Electronics Engineering and Computer Science (ICEEECS 2016)

# Accelerating the Shuffle Phase to Speed up MapReduce Systems

Rujie Yu[a], Songping Yu[b] and Nong Xiao[b]

State Key Laboratory of High Performance Computing, National University of Defense Technology, ChangSha, China

[a]yurujie2012@163.com,[b]we.isly@163.com

**Keywords:** RDMA; MapReduce; Shuffle Phase

**Abstract.** The CPU-centric traditional network protocol processing limits the utilization of network bandwidth, even with the high speed network (100Gbps); and the situation is more obvious in big data systems. The high performance network technology-Remote Direct Memory Access(RDMA), has the benefit of directly accessing remote application's memory without involving destination CPUs, broadening the performance boundary. In this paper, we build a pluggable shuffle module to boost the Map-Reduce system based on Unreliable Datagram transport of RDMA, including data fragmenting and a "Dynamic Controllably Apply and Allot" mechanism to ensure the reliability of data transmission. Experimental result shows that the performance of RDMA-based Spark is circa 16% better than that of IPoIB-based Spark.

## 1. Introduction

In the era of "Big Data", the growing demand for processing the large-scale data has spurred the development of novel systems in industry, such as Google's MapReduce[1]. The most popular framework for analysis of vast data sets in cluster machines is the MapReduce. The two top open-source Apache projects of MapReduce's implementation: Hadoop[2] and Spark[3], have attracted an ocean of researchers and engineers to contribute from industry and academia in recent years.

MapReduce is a framework for parallel processing of massive data sets. A *job* to be performed using the MapReduce framework has to be specified as two phases: the *map* phase as specified by a Map function (also called *mapper*) takes key/value pairs as input, possibly performs some computation on this input, and produces intermediate results in the form of key/value pairs; and the *reduce* phase which processes these results as specified by a Reduce function (also called *reducer*). The data from the map phase are *shuffled*, i.e., exchanged and merge-sorted, to the machines performing the reduce phase. It should be noted that the shuffle phase can itself be more time-consuming than the two others depending on network bandwidth availability and other resources. Recently, a multitude of researches have addressed this issue and seen the performance improvement of shuffling by either revising the existing implementations [4-6], or by introducing new shuffling schemes [7, 8].

In this paper, we reconstruct the data transfer mechanism of shuffle phase based on the fast network technology-Remote Direct Memory Access (RDMA) from the scratch. Commonly, in traditional network, data transfer involves packing and unpacking through user space, kernel space, and the network device. This process results in memory copy, context switch, and CPU. Therefore, when a mapper shuffles its data to reducers, the overhead of data transfer is huge. In contrast, RDMA enables CPU in one machine to remotely direct read or write the memory of another remote machine without involving the CPU overhead and the OS kernel of the remote machine. Early experiences [5, 6] with integrating RDMA into MapReduce systems mainly focused on the reliable transmission, while we strive to design with unreliable transmission. The design mainly incorporates two recipes: data fragmenting to adapt the underneath maximum trasmission unit of Ureliable Datagram transport and a "Dynamic Controllably Apply and Allot" mechanism to gain the reliability of data transmission. The experimental rersult shows that can efficiently improve the performance of MapReduce systems.

## 2. Background

RDMA hosts communicate using queue pairs (QPs); hosts create QPs consisting of a send queue and a receive queue, and post operations to these queues using the verbs API. We call the host initiating a verb the positive side and the destination host the passive side. RDMA transports are either reliable or unreliable, and either connected or unconnected (also called datagram).With reliable transports, the NIC uses acknowledgments to guarantee in-order delivery of messages. Unreliable transports do not provide this guarantee. Connected transports require one-to-one connections between QPs, whereas a datagram QP can communicate with multiple QPs. We consider one type unconnected transports in current RDMA hardware: Unreliable Datagram (UD). The other two types of connected transports are Reliable Connected (RC) and Unreliable Connected(UC).

## 3. Design and Implementation of RDMA Shuffling

**Overview**. The RDMA shuffling is integrated to MapReduce systems through the Java Native Interface (JNI) in a pluggable way, and the internals are depicted in the figure 1. On one side, the positive (send) side of RDMA shuffling is shown in the upper left of figure 1, the data blocks of mapper threads are delivered to several memory fragments from the memory pool which is directly accessed by RDMA network interface card (RNIC). The disassembly of large block from the upper (java) layer is handled by the threads in native layer. So, when RNIC can concurrently transmit as many fragments as possible to fully utilize the bandwidth of RNIC.

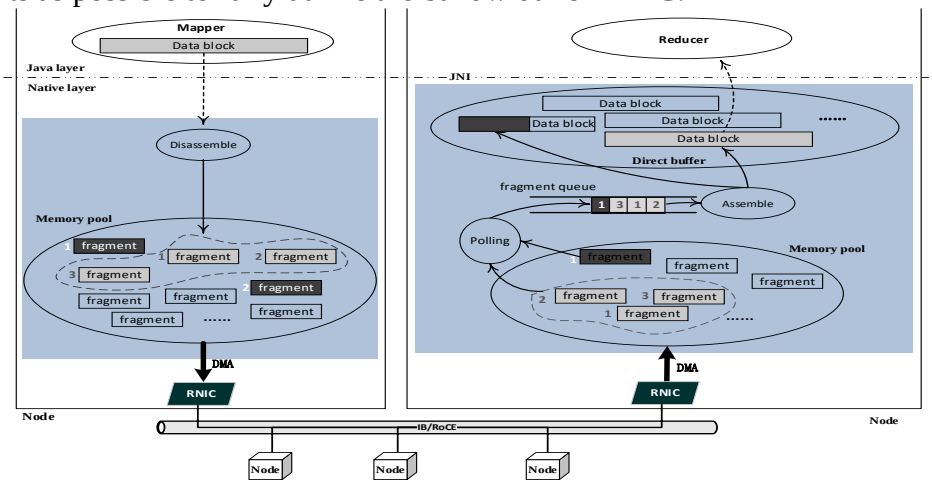


Fig. 1. The internals of RDMA-based shuffling

On the other side, the passive (receive) side of RDMA shuffling is revealed in the upper right of figure 1. In the native layer, to begin with, the polling threads detect the fragments from the memory pool accessed by RNIC at intervals and put the received fragments into a fragment queue; then the assembling threads assemble the fragments into one large data block from the queue; finally, the one large block is pushed into the reducer threads for consuming. Besides, the entire process of receiving data in accordance with the design of Staged Event-Driven Architecture (SEDA) [9]. Various number of threads can be adjusted by the parameters, in order to achieve equilibrium stages of processing a time, to achieve high throughput goals. The assembled data blocks is stored into the direct buffer directly accessed by java layer, in order to reduce the data copy overhead between the Java virtual machine and native library. This shuffling design with RDMA may provoke you to wonder: why does it need to chop the data block into small chunks since the RDMA can transfer 4 Gigabytes data under the RC mode? The next section will explain this.

**Why fragmentation?** The reason we need fragmentation mechanism is because RDMA shuffling is build based on UD mode and the maximum transmission unit of UD mode is 4096 bytes. Basically, RDMA provides three types of transmission: RC, UC and UD. While RC and UC are connection-oriented, so for each connection, both one side creates Queue Pair (QP), and establishes initiatively a connection with the remote QP before transmitting data. As the scale of MapReduce cluster increases, the number of QPs will also rises up. However, the metadata of active QPs of both

sides are cached in the small on-chip cache of the NIC, and the space of on-chip cache is a precious resource. If the number of QPs is too large, the cache miss will happen in the on-chip cache and the NIC will get the metadata of active QPs from the host via PCIe[10], resulting in performance degradation. In practice, the data shuffling in MapReduce system with hundreds nodes is frequent. If trying to shuffling data in connection-oriented mode, performance degradation is evitable. Thing is totally different for UD mode, one QP can handle all data transmission. But since UD is connectionless, the reliability of data transmission is not guaranteed. Although with the modern RDMA implementations, such as in Infiniband, and they use a lossless link layer that prevents congestion-based losses using link layer flow control [11], and bit error-based losses using link layer retransmissions [12], so UD communication can cause packet loss only due to link bit errors at only a very low probability. What's more, the retry mechanism of upper application layer can also ensure the reliable data transmission in some MepReduce systems, such as Spark. Hence, we decided to choose UD transmission mode for data shuffling. Now that we justify the shuffling design based on UD mode, then the next question is how to implement?

**Implementation.** When applications take advantage of RDMA to communication, the passive side must post a recv operation before the positive side posts a send operation. However, when the MapReduce application runs on a cluster, any node can start a data transmission connection, so it is arduous to post enough number of recv operations in advance in one node. The normal method for RC mode transmission is: Each node initially posts a recv opeartion, then re-posts a recv operation for the next data arrival after receiving data. Nevertheless, what if the passive side receives a message but has not had time to post a new recv operation before a new message? RC transmission can ensure the data retransmission, while the UD transmission is unable to do this.

To address this problem, we designed a "Dynamic Controllably Apply and Allot" (DCAA) mechanism to ensure the reliability of data transmission. The principle is that each node maintains a table that records the number of "tokens" which are allotted by other nodes. One node needs to apply for a local token allotted by remote node before post a send operation. When the token number is zero, the thread blocks until receiving the command to increase the number of tokens from the corresponding remote node. At the same time, after the token is allotted, according to the remaining number of tokens and the number of remaining fragments to predict whether or not to apply for a new token and the number of tokens, then this information will fill into the appropriate fields in the header of the sending message. Upon receiving the fragments in the passive side, it first checks whether there is a hint for applying tokens, and if there is, posting the corresponding number of recv operations, then sending a control message to the positive side to update the number of tokens. The DCAA mechanism requires to add slice 4 bytes for length and several bytes for control messages to the fragment header. The fragment header is only 24 bytes，so the extra data transmission overhead is very negligent. At the same time, the number of remaining "tokens" is usually greater than zero, so the performance degradation seldom appears due to threads blocked.

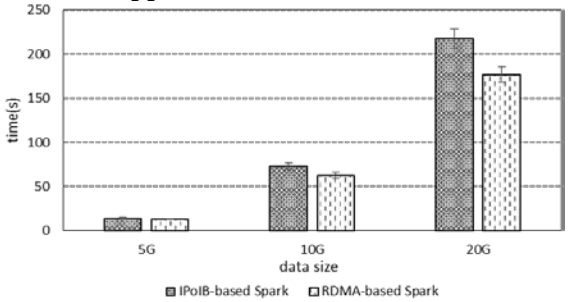


Fig. 2. The performance comparison of Grep

**Evaluation.** In order to examine the performance of the RDMA shuffle module, we integrate it into the most popular MapReduce system-Spark (version 1.6.0). We run the RDMA-based Spark with one big data benchmark-Grep on a small cluster of 4 nodes. Figure 2 shows that the performance comparison of Grep, and the performance of RDMA-based Spark is circa 16% better than that of IPoIB-based Spark.

## 4. Conclusion

The fast network technology-RDMA has broaden the boundary of big data systems due to its capability of data transmission without involving destination CPUs. In this paper, we build a pluggable shuffle module to boost the Map-Reduce system based on Unreliable Datagram transport of RDMA, including data fragmenting and a "Dynamic Controllably Apply and Allot" mechanism to ensure the reliability of data transmission. Experimental result shows that it can efficiently improve the performance of MapReduce systems.

## Acknowledgements

This work was financially supported by the is supported by the National High-Tech Research and Development Projects (863) and the National Natural Science Foundation of China under Grant Nos. 2015AA015305, 61232003, 61332003, 61202121.

## References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI), CA, December 2004.

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

[3] Apache. Apache spark. http://spark.apache.org/.

[4] Davidson and A. Or. Optimizing shuffle performance in spark. University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep, 2013.

[5] X. Lu, M. W. Rahman, N. Islam, D. Shankar, and D. K. Panda. Accelerating spark with rdma for big data processing: Early experiences. In Annual Symposium on High-Performance Interconnects (HOTI), pages 9-16. IEEE, 2014.

[6] M. Wasi-ur-Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance rdma-based design of Hadoop mapreduce over in_niband. In International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pages 1908-1917. IEEE, 2013.

[7] M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint optimization of overlapping phases in mapreduce. Performance Evaluation, 70(10):720-735, 2013.

[8] J. Ullman. Mapreduce algorithms. In Proceedings of the 2Nd IKDD Conference on Data Sciences, CODS-IKDD '15, pages 1:1-1:1, New York, NY, USA, 2015. ACM.

[9] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), Banff, Alberta, Canada, 2001.

[10] Kalia, Anuj, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. 2016 USENIX Annual Technical Conference (USENIX ATC 16). 2016.

[11] Infiniband architecture specification volume 1. https://cw.infinibandta.org/document/dl/7859.

[12] Mellanox OFED for linux user manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.2-1.0.1.pdf.