Network optimization for distributed memory file system on high performance computers

Chun-Jia Wu†, Guang-Ming Liu, Xin Liu
College of Computer, National University of Defense Technology, Changsha, Hunan Province, China
†E-mail: wuchunjiamail@163.com

On high performance computers, each compute node's demand for memory may be different. Building a distributed file system in idle memory can improve the memory utilization of the whole system. When memory takes place of disk, the Socket-based communication becomes the main bottleneck. As most of current high performance computers support RDMA, RBP (RDMA Buffer Pool) was referred to optimize the network performance of such an in-memory file system. MooseFS was adopted by RBP and deployed in TH-1A supercomputer. The experiment results showed that the RBP-based method could improve the speed of clients and the aggregate bandwidth of servers for sequential read and write significantly. For a single client, it increased the speeds of sequential read and write by a factor of 2.0~2.6. For a single server, it increased the collective bandwidths of sequential read and write by a factor of 2.0~2.4.

Keywords: high-performance computing; distributed file system; memory; RDMA; network optimization.

1. Introduction

The memory utilizations of different nodes in high performance computers are often tense or idle due to different applications. As a result, the system's overall memory resources cannot be fully utilized. This paper builds a distributed memory file system on compute nodes based on MooseFS [1] (Moose File System). At this time, the Socket-based communication becomes the main bottleneck. For this reason, this paper proposes a data transmission mechanism named RBP, which can improve the system's network performance greatly.

2. Related Work

2.1. *Overview of MooseFS*

Distributed file systems make great progresses with big data, cloud computing and high performance computing flourishing, The architecture of

GFS[2](Google File System) is significant and representative. Since GFS is not open source, MooseFS, which refers to it, is chosen. It includes four components:

(1)Master, which is the metadata management server. It maintains metadata, manages data storage server, schedules file read and write, etc.

(2)Metalogger, which is the metadata log server. It backs up Master's change log files for recovery when Master fails.

(3)Chunkserver, which is the data Storage Server. It provides data transmission and storage services for Clients under the scheduling of Master.

(4)Client. It mounts MooseFS via FUSE(File system in User Space), which allows users to use MooseFS file system as they use a local file system.

2.2. *RDMA-related researches*

Most high-performance computers interconnect compute nodes through RDMA-enabled network. Such as TH-1A, it uses a set of self-designed high-speed network supporting RDMA. Its minimum unilateral latency is as low as 1.57us, and the unidirectional bandwidth is up to 6.34GB/s[3].

Meanwhile, distributed systems often requires communication to provide higher bandwidth and lower delay. Thus, researchers have done a lot of researches on improving the network performance through RDMA. For example, NS Islam, MW Rahman, et al. improve the write performance of HDFS (Hadoop Distributed File System) by adding a Java adapter to the HDFS client, so that it can use Infiniband network through the functions of UCR (Unified Communications Runtime) Library in [4]. Christopher Mitchell, Yifeng Geng devises a distributed memory key-value storage named Pilaf, which implements a get operation based on RDMA to deal with a large number of read requests in the key-value storage in [5].

3. Socket-Based Communication in MooseFS

When Client processing a read/write request, Client writes the request information into a REQ and allocates a data buffer. Then the REQ is stored in Request Queue. Read Worker takes a REQ and sends its request information to the Chunkserver over Socket. Read Server receives the request, reads the requested data and the data as well as the status back to Client over Socket. When processing a write request, Client uses asynchronous write. It writes data directly to the CB allocated from local Write Cache and returns the status. Then Write Worker writes the allocated CBs to the Chunksever over Socket, either.

According to the analysis above, there are two steps related to the actual data operations: one is the local operations of Chunkserver; the other is the transmission between Client and Chunkserver. We carried out a comparing

experiment, in which MooseFS is deployed on disks and in memory respectively. The experiment result shows that although the server write speed of memory-based storage increases several times, but the system write speed is not significantly improved due to the Socket-based communication .

4. RBP-based Optimization Method

4.1. *RDMA Buffer Pool*

The principle of RBP is to pre-register one or more areas of memory to support RDMA operations, and divides the registered areas into a number of blocks of different sizes. These blocks are referred as RBB(RDMA Buffer Block). According to different purposes, RBBs are organized into different RBPs, then the RBPs can provide high performance data transmission service with a set of dedicated APIs.

RBP consists of a double-linked list of RBBs, an array for managing the registered memory areas, counters, mutually exclusive variable, etc.

RBB consists of description area, request area and data area. The description area is to provide necessary information for RDMA communication. The request area is to provide control messages. The data area is to provide space in the registered memory area.

When using RBB for RDMA communication, a pair of RBBs is required at both ends of the communication. In order to use RBP simply and flexibly, a set of APIs has been devised, with which RBP can be explicitly or implicitly used. When RBP is used explicitly, users obtain all the RBBs of RBP by allocation after RBP is created and manage these RBBs by themselves. When RBP is used implicitly, users allocate RBBs from RBP on demand and release them after using. A dedicated management module is needed to allocate and release RBBs.

4.2. *Read optimization*

4.2.1. *Dual RBPs*

Each read request is allocated a data buffer, if the data buffer of an RBB is used as the data buffer of a read request, data can be read directly from Chunkserver over RDMA. For this reason, a dedicated Req RBP in which the size of RBBs equals to the chunk size, is set to provide data buffers for read requests. But the RBBs in Req RBP need larger data buffer, the number of RBB is limited. So Client also sets a Read RBP to provide temporary data buffers. Req RBP and Read RBP complement each other. To cooperate with the two RBPs of Client, Chunkserver also sets a Read RBP. The sizes of Read RBPs of Client and

Chunkserver are both 64MB, and the sizes of RBBs in them are both 64KB. In addition, all RBPs serving for read are implicitly used.

4.2.2. *Multi-channels*

To take full use of Req RBP and Read RBP of Client, a control strategy is designed according to their purposes. When the size of a read request exceeds 1 MB, the read request allocates a RBB from Req RBP first. If the allocation fails, the read request allocates a pieces of memory as its data buffer. And Read RBP is used to provide temporary data buffers for the read requests which cannot allocate a RBB from Req RBP. Besides, RDMA performs worse than Socket in the transmission of non-continuous small data. Therefore, Chunkserver decides which communication methods is used in the provision of data. Only when data to be transmitted is less than 32KB, it uses Socket. In other cases, it uses RDMA.

Based on the control strategy above, there may be 3 channels when processing read requests. As shown in Figure 1(a), channel (1), (2) reads data over RDMA, channel (1) uses Req RBP to receive data for Client, channel (2) uses Read RBP to receive data for Client; channel(3) reads data over Socket.

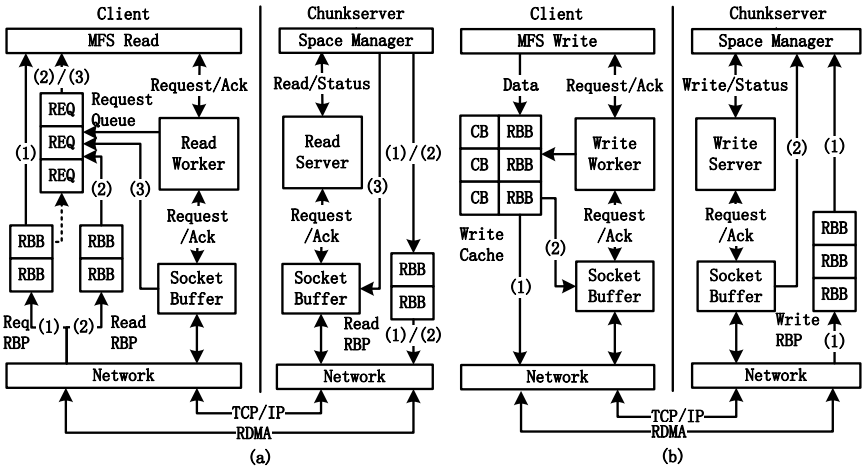

Fig. 1 RBP-based Read/Write Flow in MooseFS

4.3. *Write optimization*

Client sets a Write RBP, which combines explicitly with Write Cache to reduce data copy and save space. When Write Cache is initialized, each CB binds a RBB of Write RBP. Thereafter, Write RBP is managed by Write Cache. To cooperate with Write RBP of Client, Chunkserver also sets a Write RBP as a data buffer pool. And the size of its RBBs equals to that of Client.

For the same consideration in Section 4.2.2, write also supports multi-channels. The difference is that it's Client to decide which communication method to use before writing data to ChunkServer. As shown in Figure 1 (b), channel (1) writes data over RDMA , channel (2) writes data over Socket.

5. Performance Evaluation

5.1. *Experiment environment*

Based on TH-1A supercomputer, following experiments are conducted to validate the performance improvement brought by the RBP-based network optimization method. Choosing MooseFS 3.0.73 to be the contrast version, the Socket-based version and RBP-based version are abbreviated as Socket and RBP. One compute node acts as Master, four compute nodes with FUSE module mount MooseFS. Another compute node with 48GB of memory acts as Chunkserver and provides 40GB of storage space. The test tool chooses IOR with version 2.10.1. The size of test files are set to be 2GB, and the block size of them ranges from 16KB to 4MB. Direct I/O mode of Client is enabled.

5.2. *Results and analysis*

Comparison experiments of Client are conducted on a single Client. As shown in Figure 2, in the situation of the same file block size and the same number of processes, the sequential read speed of improved system can speed up to 2.02 times that of contrast system and the sequential write speed can speed up to 2.63 times. In addition, when the number of processes ranging from 4 to 8, there is no significant increase for the contrast system, indicating that it is close to the upper limit through Socket communication.

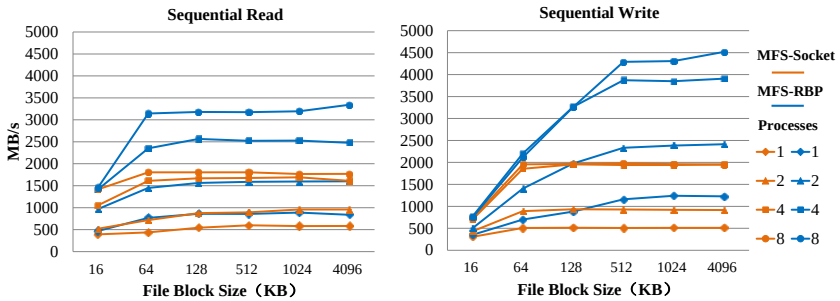

Fig. 2 Comparisons of Sequential Read/Write Speed of a single Client over Socket or RBP

Comparison experiments of Server are conducted on a single Chunkserver with 4 Clients. Each client uses a single process to read or write concurrently to test the aggregate bandwidth that a single server can provide. The results of these experiments are shown in Figure 3. The aggregate sequential read

bandwidth that the improved system provide can be up to 2.04 times that of the contrast system. The aggregate sequential write bandwidth can be up to 2.35 times and its maximum value is as high as 4.42GB/s, which accounts for nearly 70% of the maximum unidirectional bandwidth between compute nodes.

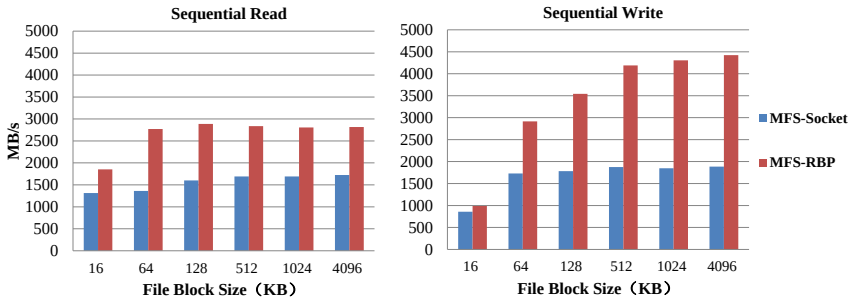

Fig. 3 Comparisons of Sequential Read/Write Bandwidth of a single Server over Socket or RBP

6. Conclusion and Future Work

This paper proposes a data transmission mechanism RBP based on RDMA. RBP can be used to improve the data transmission performance of Socket-based distributed file systems without changing the original control flow. The future work is to combine the techniques of data prefetching and write combing to make the RBP-based network optimization method have more comprehensive performance and wider application prospect.

References

1. Bai S, Wu H. The Performance Study on Several Distributed File Systems[C]// International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery. IEEE, 2011:226-229.
2. Ghemawat S, Gobioff H, Leung S T. The Google file system[J]. Acm Sigops Operating Systems Review, 2003, 37(5):29-43.
3. Xie Min, Lu Yutong, Liu Lu, et al. Implementation and Evaluation of Network Interface and Message Passing Services for TianHe-1A Supercomputer[C] //Proc of the 2011 IEEE Symposisum on High Performance Interconnects. Piscataway, NJ: IEEE, 2011:78-86.
4. Islam N S, Rahman M W, Jose J, et al. High performance RDMA-based design of HDFS over InfiniBand[C] //Proc of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE Computer Society, 2012:1-12.
5. Mitchell C, Geng Y, Li J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store[C] //Proc of USENIX ATC '13. Berkeley, CA: USENIX, 2013:103-114.