HLD of LUSTRE ADIO COLLECTIVE WRITE DRIVER (v6)

LiuYing

July 6rd, 2008

1 Introduction

Numerous studies have shown that many scientific applications need to access a large number of independent and small pieces of data from file. The I/O performance suffers considerably if applications access data by making many small I/O requests. To improve the parallel I/O performance, people try to collect the small I/O request into some big ones. A simple test was performed on ORNL Jaguar with Lustre file system in IOR benchmark. Shown in Table1, the performance of big size I/O is much better than the small one for the same size file.

Table 1: The performance difference between small and big size I/O

block size	proc_num	B.W.(MB/s)	open(s)	write(s)	close(s)
256KB	256	604.23	0.086969	0.013279	0.005586
1MB	64	1955.86	0.035173	0.006083	0.001563

MPI-IO ADIO is an abstract-device interface for implementing portable parallel-I/O interfaces. It provides collective I/O mode to merge the requests of the different parallel processes and serve the merged requests. It has been applied on many parallel file systems, including PVFS, PANFS, PIOFS, NFS and so on. In this HLD, ADIO collective write driver for Lustre file system is introduced. Its perceived advantages are:

- to saturate the network and disk IO with less RPC, and
- to avoid unnecessary extent lock conflicts

Understanding MPIIO and ADIO is assumed.

2 File-contiguous vs. Stripe-contiguous

Since Lustre ADIO driver is used for transfering Application IO pattern to the pattern Lustre prefer. Usually, there are two choices file-contiguous and stripe-contiguous. As shown in Figure1,

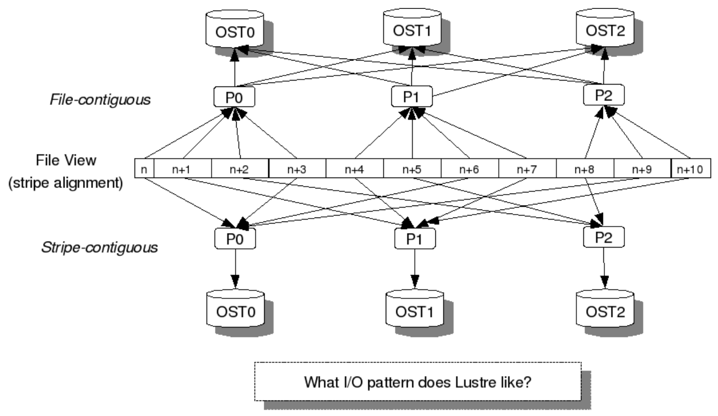

Figure 1: I/O patterns for Lustre

- File-contiguous: Data are written in file offset sequence. One possible distribution is given in Figure1 that p0 writes $[n \sim (n+3)]$, p1 writes $[(n+4) \sim (n+7)]$ and p2 writes the remains. In this pattern, each process might access all the OSTs.
- Stripe-contiguous: Data belonging to the same OST are collected and then redistributed to the I/O clients. Here, I/O client means the process who performs I/O to file. Also, in Figure1, p0 writes $[n,(n+3),(n+6),(n+9)]$ to OST0, p1 writes $[(n+1),(n+4),(n+7),(n+10)]$ to OST1 and p2 writes the left to OST2. Usually, in this pattern, each I/O client only accesses one OST and one OST can be accessed by one or more I/O clients.

Which one does Lustre prefer? The testing results are shown in Figure2. In the test, stripe_count=72, nprocs=1024 and stripe_size=4M. Blocksize was changed from 4M to 64M exponentially. The results show that Lustre can achieve better performance in stripe-contiguous pattern than file-contiguous pattern, due to less extent lock conflicts.

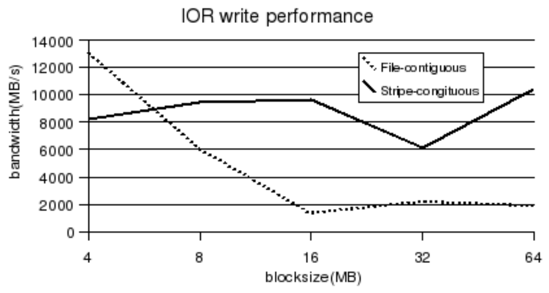

Figure 2: performance results of file-contiguous vs. stripe-contiguous

According to the results, Lustre ADIO driver should provide a good manner to convert the application I/O patterns to Lustre stripe-contiguous pattern by balanced OST load and efficient RPC between clients and servers, in low overhead.

3 Definitions

CO: In stripe-contiguous IO pattern, each OST will be accessed by a group of clients. (Note: each client will only access 1 OST). CO(client OST ratio) means the number of clients for each group, the default CO is 1, which means the data belongs to one OST will be reorganized to 1 client, then write.

Data sieving: MPI-IO provides data-sieving technique for *single process* to improve I/O performance in independent I/O mode. It is applied when a lot of small discontiguous data segments are written. It is actually a read-modify-write process. First the big contiguous write buffer is allocated to cover all the data segments, and then lock the whole area and read it from the file, then put all the data segments to the write buffer, finally write to the file with the whole contiguous write buffer.

4 Requirements

Data redistribution reorganize the I/O requests from the application into the pattern lustre prefer(stripe contiguous).

Low Overhead There are two kinds of overhead. One comes from collective communication[section7.2.1] and the other is file locking from read-modify-write in local I/O phase[section7.2.2]. They all degrade the system performance very much, so overhead reduction is very important.

5 Functional specification

5.1 Data redistribution

The aim of data redistribution is to generate stripe-contiguous pattern. To meet it, several subfunctions should be involved.

5.1.1 Stripe alignment

Obviously, stripe-contiguous pattern needs stripe alignment, which has been proved to be effective to reduce unnecessary extent lock conflicts and get adequate utilization of network& disk I/O with less RPC.

5.1.2 Application I/O pattern identification

Before producing stripe-contiguous pattern, Lustre ADIO driver should identify whether the current application access pattern can benefit from collective I/O mode. In section6, three typical use cases are discussed.

5.1.3 Data reorganization

Once the application access pattern passes the check, Lustre ADIO driver will collect data according to which OST they are located in, and then redistribute the data to the I/O clients. During this procedure, some policies to keep load balancing between I/O clients and OSTs should be adopted, as follows.

- Each OST should be accessed by almost the same number($\leq CO$) of I/O clients each time. However, sometimes more I/O clients will bring higher bandwidth, so there is a trade-off.
- For the ($\leq CO$) I/O clients to each OST, each I/O client should send almost the same amount of I/O load each time. The data should be aligned with stripe size.

6 Use cases

Three use cases are given to explain how Lustre ADIO driver will reorganize the application IO data. 3.

- Case1: The I/O request size is smaller than stripe size and the data are contiguous.

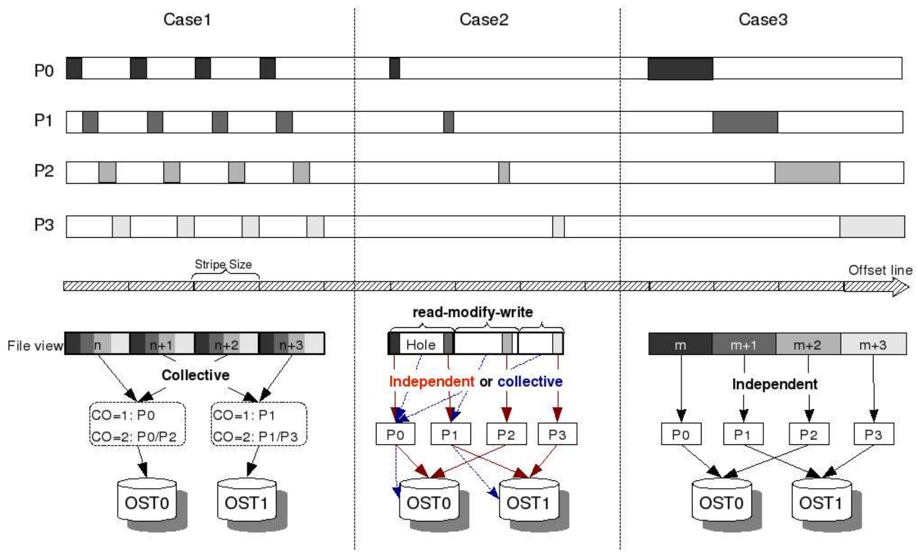

Figure 3: I/O pattern identification for three use cases

  - ADIO driver redistributes the data to the IO clients according to CO as Figure33,
    - When CO = 1, data will be reorganized to 2 IO process (P0–>OST0 and P1–>OST1), then write.
    - When CO = 2, data will be reorganized to 4 IO process.(P0, P2 ->OST0 and P1,P3->OST1), then write.

- Case2: The I/O request size is smaller than stripe size but the data are non-contiguous.
  - Usually, after the data being reorganized into IO clients, which will be writen by stripe_size. If these data are non-contiguous, read-modify-write might happened at this time. End user could provide the hints to inidicate this. Read-modify-write will not be done in default case. Case 2 in Figure 3 indicate the process of this process.

- Case3: The I/O request size is no smaller than stripe size.
  - When the IO request size is bigger than stripe size, the data should still be reorganized into stripe-contiguous pattern according to CO.

7 Logic specification

Two-phase I/O is the most important part in ADIO collective driver, including communication phase and I/O phase[2]. The diagram in Figure4 gives a whole perspective of Lustre ADIO driver in detail. First, the application I/O patten is checked whether it can benefit from collective I/O. If so, these I/O requests are converted into stripe-contiguous pattern, otherwise the data are written to the file in contiguous or strided I/O mode. Meanwhile, we optimize collective communication in communication phase and to reduce unnecessary locking overhead in I/O phase, respectively.

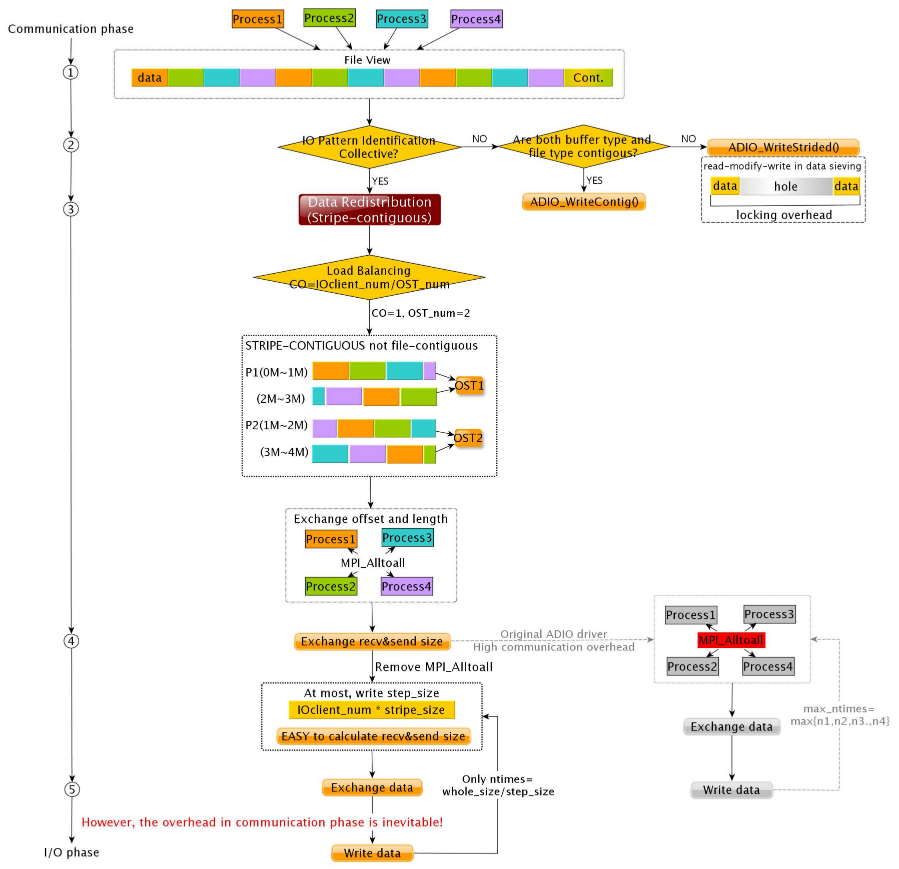
Communication phase
Process1
Process2
Process3
Process4
File View
data
Cont.
IO Pattern Identification Collective?
NO
Are both buffer type and file type contigous?
NO
ADIO_WriteStrided()
read-modify-write in data sieving
data
hole
data
locking overhead
YES
YES
ADIO_WriteContig()
Data Redistribution (Stripe-contiguous)
Load Balancing CO=IOclient_num/OST_num
CO=1, OST_num=2
STRIPE-CONTIGUOUS not file-contiguous
P1(0M~1M)
(2M~3M)
OST1
P2(1M~2M)
(3M~4M)
OST2
Exchange offset and length
Process1
Process3
MPI_Alltoall
Process2
Process4
Exchange recv&send size
Original ADIO driver
High communication overhead
Process1
Process3
MPI_Alltoall
Process2
Process4
Remove MPI_Alltoall
At most, write step_size
IOclient_num * stripe_size
EASY to calculate recv&send size
Exchange data
max_ntimes=max{n1,n2,n3,,n4}
Exchange data
Only ntimes=whole_size/step_size
Write data
However, the overhead in communication phase is inevitable!
I/O phase
Write data

Figure 4: Diagram of Lustre ADIO collective write driver

7.1 Data redistribution

7.1.1 Application I/O pattern identification

After each process analyzes its own I/O request, each process determines whether the current application access pattern are suitable for collective I/O in step2 of communication phase in Figure4. The algorithm for I/O pattern identification is described below.

```
int ADIOI_LUSTRE_Docollect(...)
{
    ......
    /* Calculate the total_req_size and total_access_count */
    ...
    /* Estimate average I/O request size */
    avg_req_size = total_req_size / total_access_count;
    /* Calculate total hole size */
    total_hole_size = whole_access_range - total_req_size;
    ......
    /* I/O patten identification */
    if (avg_req_size > stripe_size) {
        /* Use Case3: avg_req_size > stripe_size */
        docollect = 0;
    } else if (total_hole_size * hint_N > whole_access_range) {
        /* Use Case2: big hole */
        docollect = 0;
    } else if (total_hole_size == 0) {
        /* Use Case1: compared with CO */
        if (avg_req_size * CO < stripe_size) {
            docollect = 1;
        } else {
            procs_per_stripe = (stripe_size - 1) /
                                   avg_req_size + 1;
            stripes_per_ost = (avg_req_size * nprocs - 1) /
                                  (stripe_size * stripe_count) + 1;
            if (procs_per_stripe_size * stripes_per_ost > CO)
                docollect = 1;
            else
                docollect = 0;
        }
    }
    ......
    return docollect;
}
```

7.1.2 Data reorganization

It takes place in step3 of communication phase in Figure4. In this step, the key point is *how to redistribute collective data to the I/O clients to produce stripe-contiguous I/O pattern*. The algorithm is

```
/* IO client number */
avail_nprocs = ADIOI_MIN(stripe_count * CO, nprocs);
/* Redistribution and stripe alignment */
```

```
    rank_index = (offset / stripe_size) % avail_nprocs;
```

In Figure4, if $OST\_num = 2$ and $CO\_hint = 1$, there are only 2 I/O clients out of 4 processes. Then, all the data belonging to OST1 will be written by p1, and those belonging to OST2 will be written by p2, same to the description of stripe-contiguous pattern in Figure1.

7.2 Overhead reduction

7.2.1 Collective communication

Collective communication is one of the most important communication types in MPI [4]. It is used to transmit data among all processes simultaneously in a group specified by an intercommunication object. However, complete exchange from all members to all members produces n^2 communication operations, which have a great impact on the performance. So, to reduce this kind of overhead, we try to avoid or simplify collective communication.

In step4 of communication phase shown in Figure4, because of the merged big size I/O, before recv/send data among the processes, it always takes each process more than one times($ntimes$) of communications to make sure of which process and what size data it will receive from or send to.

In the original driver, ntimes depends on the maximum of all the processes due to the different file portion, and MPI_Alltoall() is called in each time communication, so that performance would suffer from unnecessary communication overhead, as shown in the bottom right of Figure4.

To improve that, we not only reduces communication times, but also simplify communication operations by removing MPI_Alltoall. Because the data redistributed to each I/O client are stripe-contiguous not file-contiguous, all the processes can complete their data size exchanging in the same file portion between the maximum end offset and the minimum start offset. And because all the I/O clients write $IOclient\_num \times stripe\_size$ size I/O at most each time, the data size to be sent and received is easy to calculate without MPI_Alltoall(), and all the processes have the same $ntimes = \frac{max\_end\_offset-(min\_start\_offset-min\_start\_offset\%stripe\_size)}{IOclient\_num \times stripe\_size} + 1$.

7.2.2 Locking in data sieving

Read-modify-write in data-sieving can keep the data in the holes from destruction for single process in strided I/O mode. But sometimes, it would cause high locking overhead and extra I/O operations. To avoid this, a read-modify-write won't be adopted except that the current I/O pattern has been validated or the users enable it.

In I/O phase, there are four kinds of I/O writing from the buffer to the file, according to their contiguous/non-contiguous types. When file type is contiguous, the operation is relative easy. Each contiguous data block is copied to the temporary write buffer, and then write the temporary buffer to the file. Here, the temporary write buffer is set to stripe size by default.

But, if file type is non-contiguous, a read-modify-write would be triggered. To avoid this, if the buffer size is no larger than the contiguous file portion (fwr_size), the data will be written to the file directly. Otherwise, if data sieving is disabled, even though there is a hole, the data will be written to the file separately in independent I/O mode. If not, a read-modify-write will be adopted.

8 Performance results

We compared IOR write performance of Lustre ADIO driver(collective/independent) and POSIX on Jaguar with 1024 clients, 72 OSTs and CO=1. Stripe size is 1M and the blocksize was set from 1K to 64M exponentially. For convenience, write bandwidth and write time were converted into log10(). The results are shown in Figure5.

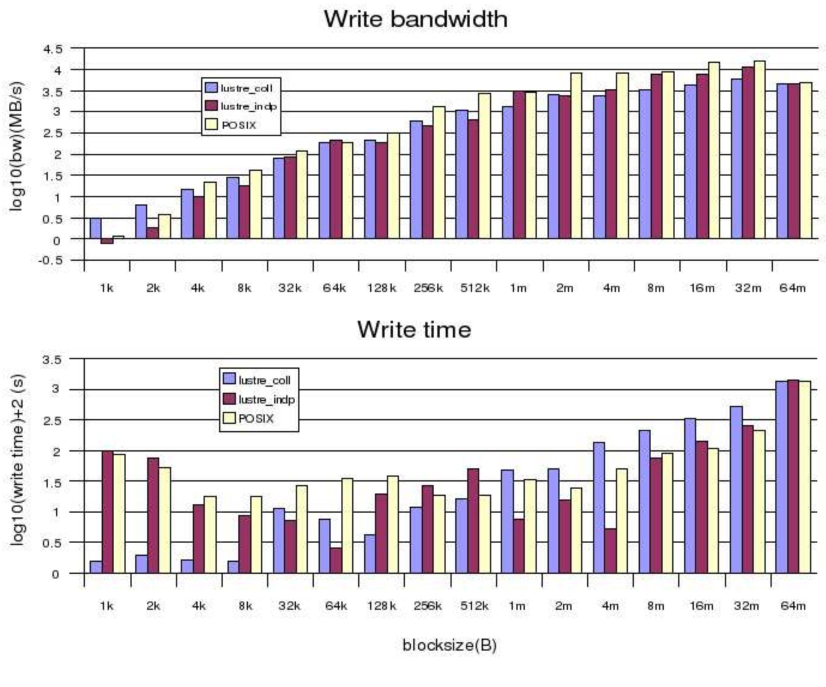

Figure 5: IOR write performance results

When blocksize was smaller than stripe size, the performance of collective write was acceptable. But with the growing of blocksize, its write time (composed by communication time and I/O time) became larger and larger, and its performance became much worse than independent I/O mode and POSIX.

The overhead analysis of Lustre ADIO collective write driver (blocksize=1K and 2K) is shown in Figure6. It shows that I/O phase(writing data) cost no more than 35% overhead and exchanging data cost more than 40% overhead.

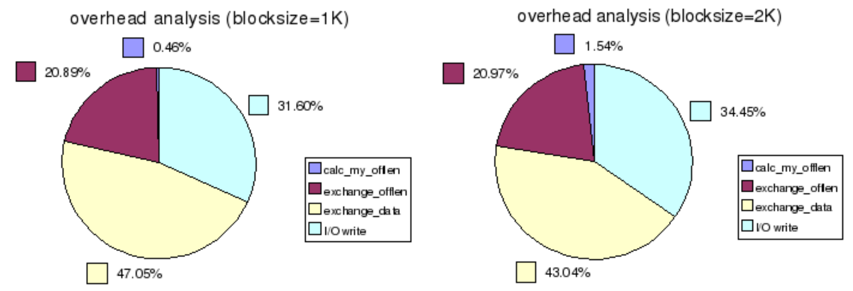

Figure 6: Overhead analysis (blocksize=1K and 2K)

Although Figure2 proved that stripe-contiguous was more suitable for Lustre than file-contiguous pattern, the collective communication overhead is hard to avoid.

We set CO=1 in this test, which means only one I/O client for each OST all the time. As mentioned in section5.1.3, more I/O clients will bring higher bandwidth sometimes. We will test it later.

9 Future improvements

9.1 I/O features

Some I/O optimization features will be exported to Lustre, including lockless I/O, disabling extending extent lock, read ahead, and so on. [3]

9.2 Collective read

In this HLD, we focus on collective write only. We will investigate collective read and other collective operations in the near future.

- When the big size I/O requests come, the data will be broken into several stripe-size segments and redistributed to the I/O clients.

10 Alternatives

The design idea in this document is still in the validation. If the final experiment results show the design can't improve the performance greatly, or the optimization requirements of the scientific applications changes, this HLD will be modified soon and the new ideas will be proposed.

11 Focus for inspections

- Data redistribution
- Overhead reduction

12 Reference:

1. MPI-IO documents for ROMIO and ADIO
2. Thakur, R. Gropp, W. Lusk, E. "Data sieving and collective I/O in ROMIO". In the Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, 1999. Frontiers '99.
3. "HLD of exporting I/O features" by WangDi
4. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, "MPI: The complete reference". The MIT Press