Lustre Technical White Paper

Lustre ADIO collective write driver

Author	**Date**	**Description of Document Change**	**Client Approval By**	**Client Approval Date**
LiuYing	April 29, 2008	Create the document		
LiuYing	May 28, 2008	Revise per HLD comment		
LiuYing	June 10, 2008	Add the test results		
LiuYing	July 10, 2008	Add the diagram		
LiuYing	July 11, 2008	Update the test results		
LiuYing	July 21, 2008	Revise per Nirant’s comment		
LiuYing	July 30, 2008	Add the description of the hints		
LiuYing	July 30, 2008	Simplify application I/O pattern identification		
LiuYing	Aug 25, 2008	Update FLASH IO results		
LiuYing	Sep 26, 2008	Add the Chimera results		

1. Problem Statement

Numerous studies have shown that many scientific applications need to access a large number of 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, the small I/O requests are collected into fewer number of larger size requests.

MPI-IO ADIO[1] is an abstract-device interface for implementing portable parallel-I/O interfaces. It provides collective I/O[2] mode to merge the requests of the different parallel processes and serve the merged requests. It has been implemented on many parallel file systems, including PVFS, PANFS, PIOFS, NFS and so on. In this report, Lustre ADIO collective write driver is introduced. It has two perceived advantages, including

- to saturate the network and disk IO with fewer RPCs, and
- to avoid unnecessary extent lock conflicts.

2. Approach

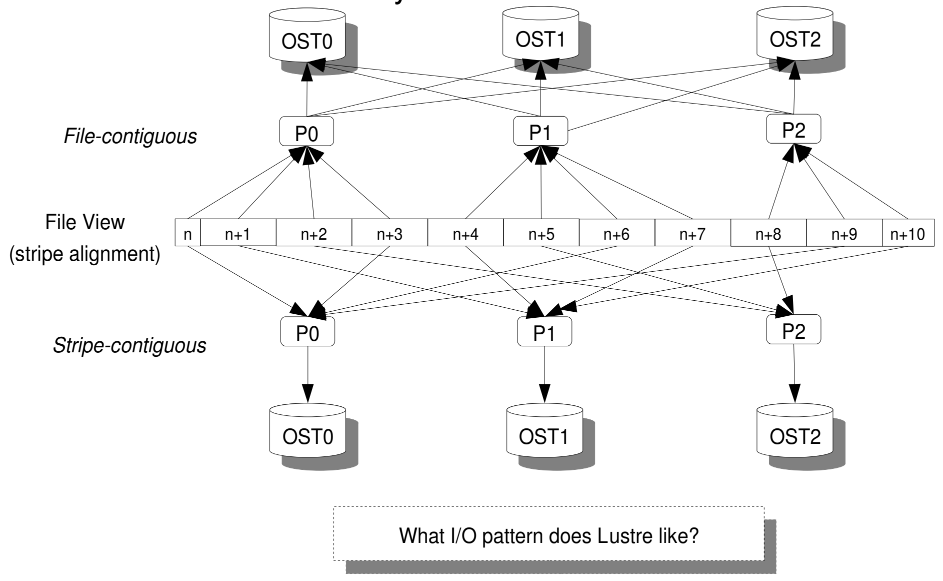
The main purpose of Lustre ADIO collective write driver is to provide an efficient way to convert the application I/O patterns to the pattern Lustre prefers with low overhead.

2.1. Lustre IO pattern

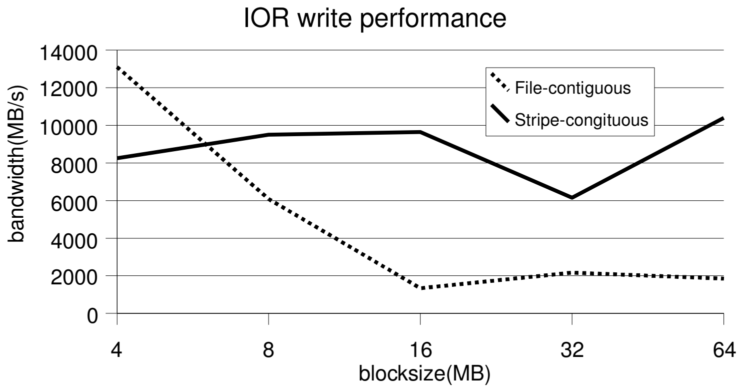
What I/O pattern does Lustre like? There are two choices, file-contiguous and stripe-contiguous, as shown in Figure1(a).

- File-contiguous: Data is written in file offset sequence. One possible distribution is given in Figure1(a) that p0 writes n~(n+3) , p1 writes (n+4)~(n+7) and p2 writes the remaining. In this pattern, each process might access all the OSTs.
- Stripe-contiguous: Data belonging to the same OST is collected and then redistributed to the I/O clients. Here, I/O client means the process which performs I/O to file. In Figure1(a), p0 writes n, (n+3), (n+6) and (n+9) to OST0, p1 writes (n+1), (n+4), (n+7) and (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.



(a) stripe-contiguous pattern



(b) performance comparison

Figure 1. Lustre I/O pattern

The comparison testing results between file-contiguous and stripe-contiguous patterns are shown in Figure1(b). The test was performed on the ORNL Jaguar system. In the test, ost_num=72, nprocs=1024 and stripe_size=4MB. Blocksize was changed from 4MB to 64MB exponentially. The results show that Lustre can achieve better performance in stripe-contiguous pattern than file-contiguous pattern due to fewer extent lock conflicts.

2.2. Definitions

For good understanding, we give some definitions here.

CO: In stripe-contiguous I/O pattern, each OST will be accessed by a group of clients.(Note: each client will only access 1 OST). CO(**C**lient **O**ST ratio) is the max. number of clients for each OST. The default CO is 1, which means the data belongs to one OST will be reorganized to 1 client, then be written.

Data sieving: MPI-IO provides data-sieving technique for single process to improve I/O performance in independent I/O mode. It uses read-modify-write to avoid destroying the data already presented in the holes between contiguous data segments, and hopefully does one I/O in the MPI-IO layer. When a read-modify-write happens, the portion to be accessed is locked and read out to the temporary write buffer, then the temporary buffer is overwritten by the user data, and at last the temporary buffer is written to the file.

2.3. Requirements

Lustre collective write driver should satisfy two requirements, including data redistribution and low overhead.

- Data redistribution

  Reorganize the I/O requests from application into the pattern Lustre prefers (stripe-contiguous).

- Low overhead

  There are two kinds of overhead. One comes from collective communication and the other comes from read-modify-write in local I/O phase. They all degrade the system performance very much.

2.4. Optimizations

Figure 2 depicts the working of the Lustre ADIO collective write driver. Two-phase I/O is shown, involving communication phase and I/O phase[2]. First, the application I/O pattern is checked whether it can benefit from collective I/O. If so, these I/O requests are converted into stripe-contiguous pattern, otherwise they are written by ADIO_WriteStrided(). Meanwhile, we optimize collective communication in communication phase and avoid unnecessary read-modify-write during data sieving in local I/O phase, respectively.

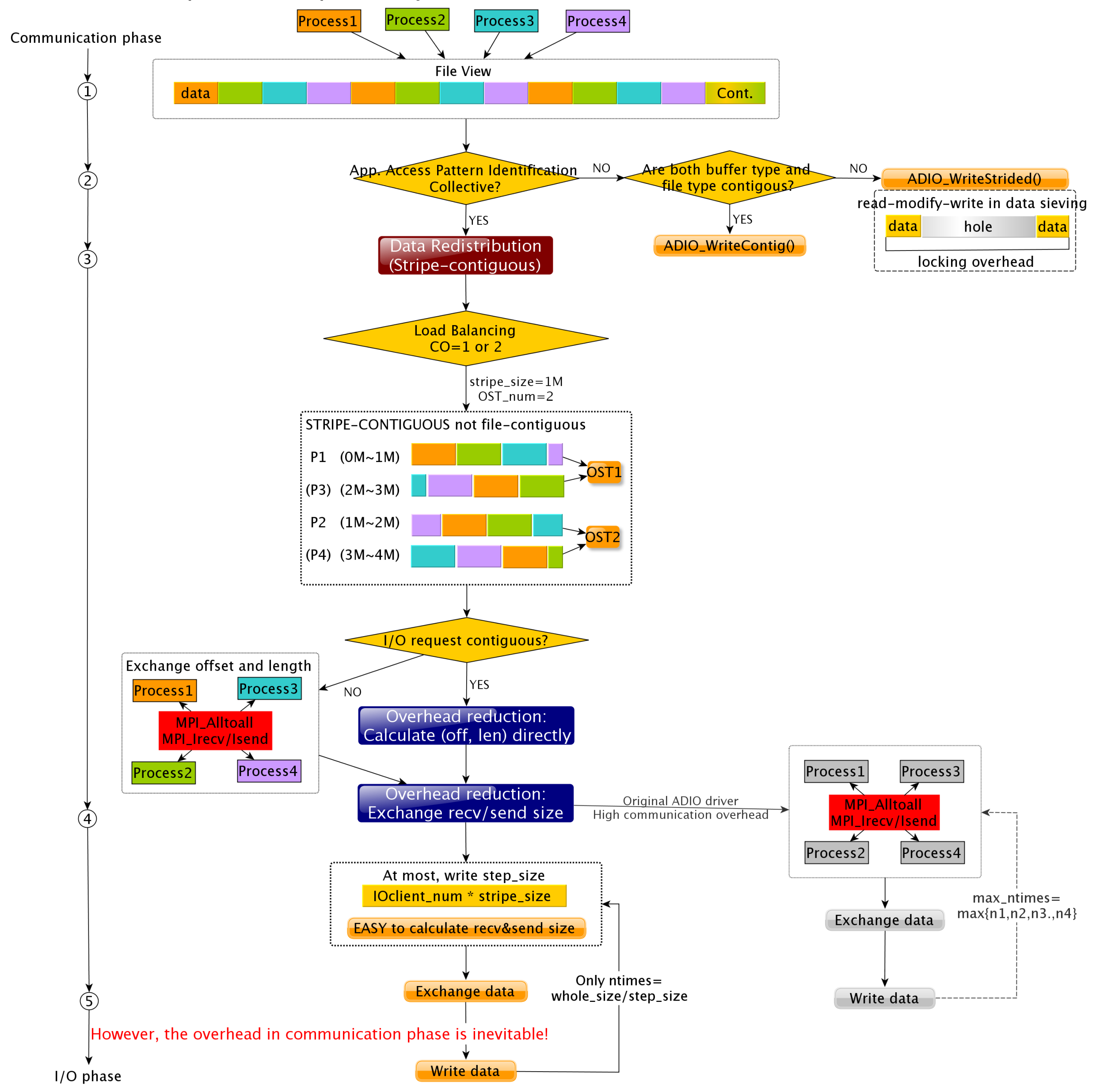

Communication phase
Process1
Process2
Process3
Process4
File View
data
Cont.
App. Access Pattern Identification Collective?
NO
Are both buffer type and file type contigous?
NO
ADIO_WriteStrided()
read-modify-write in data sieving
data
hole
data
locking overhead
YES
ADIO_WriteContig()
YES
Data Redistribution (Stripe-contiguous)
Load Balancing CO=1 or 2
stripe_size=1M OST_num=2
STRIPE-CONTIGUOUS not file-contiguous
P1 (0M~1M)
(P3) (2M~3M)
P2 (1M~2M)
(P4) (3M~4M)
OST1
OST2
I/O request contiguous?
Exchange offset and length
Process1
Process3
MPI_Alltoall MPI_Irecv/Isend
Process2
Process4
NO
YES
Overhead reduction: Calculate (off, len) directly
Overhead reduction: Exchange recv/send size
Original ADIO driver
High communication overhead
Process1
Process3
MPI_Alltoall MPI_Irecv/Isend
Process2
Process4
max_ntimes= max{n1,n2,n3.,n4}
Exchange data
Write data
At most, write step_size
IOclient_num * stripe_size
EASY to calculate recv&send size
Exchange data
Only ntimes= whole_size/step_size
However, the overhead in communication phase is inevitable!
Write data
I/O phase
1
2
3
4
5

Figure 2. Diagram of Lustre ADIO collective write driver

1. Data redistribution

   The aim of data redistribution is to generate stripe-contiguous pattern. It consists of three sub functions.

a) Stripe alignment

Stripe alignment is a precondition of stripe-contiguous pattern. It has been proven to be effective to reduce unnecessary extent lock conflicts and get adequate utilization of network and disk I/O with fewer RPC's.

b) 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. We decide this according to the request size, no matter if the data is interleaved or not. Here, we determine the request size is small or big according to the hint "*big_req_size*". (Please refer to Appendix for more details.) Two typical use cases are discussed in Figure3.

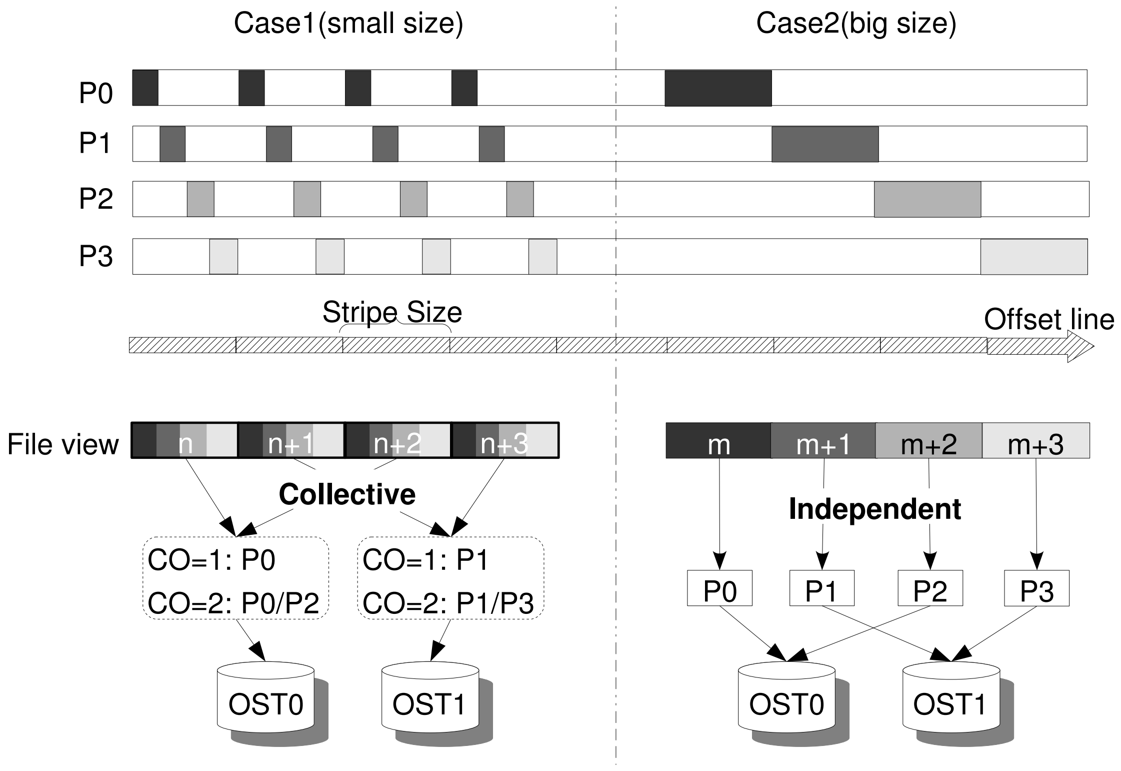


Figure 3. Application I/O pattern identification

- Case1: small size

  This case usually can benefit from collective I/O mode. For this case, the driver collects the data and redistributes them to the IO clients as shown in

Figure3.

i. When CO = 1, data will be reorganized to 2 IO clients(P0–>OST0 and P1–>OST1), then be written.

ii. When CO = 2, data will be reorganized to 4 IO clients(P0, P2 ->OST0 and P1,P3->OST1), then be written.

- Case2: big size

  Since big size requests can achieve good parallel I/O performance, we don't need collect them any more, so independent IO mode is recommended.

c) 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 the reorganization, some policies to keep the load balanced between I/O clients and OSTs should be adopted, as follows.

- Each OST should be accessed by almost the same number(<=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 I/O clients to each OST, each I/O client should send almost the same amount of I/O load each time.

2. Overhead reduction

a) Data sieving

MPI-IO provides data-sieving technique for single process to improve I/O performance in independent I/O mode. It uses read-modify-write by default to keep the data in the holes from destruction for a single process. But sometimes, it would cause high locking overhead and extra I/O operations. To avoid this, a read-modify-write won't be adopted unless we enable it. We provide hint "*ds_in_coll*" to control it. (Please refer to Appendix for more details.)

b) MPI Collective communication

Collective communication is one of the most important communication types in MPI[3]. 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, such as MPI_Alltoall(), produces n^2 communication operations, which have a great impact on the performance.

There are two places calling MPI_Alltoall in original ADIO driver. We make optimization for both of them.

i) ADIOI_LUSTRE_Calc_others_req()

In this function, each process calculates which process will access its own I/O request and then sends the access information to that process. MPI_Alltoall is used to exchange the access information among the processes. During the calculation, if each request IO size is same and the request data is contiguous, the access information(offset and length) can be calculated easily, without MPI_Alltoall(). Here, we judge the size and continuity by hints "*same_io_size"* and "*contiguous_data"*. (Please refer to Appendix for more details.)

ii) ADIOI_LUSTRE_Exch_and_write()

In this function, MPI_Alltoall is used to exchange recv/send size before IO data communication. Before optimization, each process may have different file portion, so the communication times depend on the maximum of all the processes, and MPI_Alltoall called each time makes performance suffer from unnecessary communication overhead.

As shown in the bottom right of Figure2, we optimize it according to the following criterias.

- Since the data redistributed to each I/O client is stripe-contiguous not file-contiguous, all the processes can complete their data size exchange in the same file portion.
- Through the previous calculation, each process knows clearly about which process it will receive data from and which process it will send

data to, so they don't need MPI_Alltoall() to exchange offset and length any more.

- And because all the I/O clients write $IOclientnum \times stripesize$ size I/O at most in each communication, the whole file portion can be accessed in $ntimes = \frac{(maxendoffset - minstartoffset)}{(IOclientnum \times stripesize)} + 1$

  Here, *minstartoffset* is aligned with stripe size.

3. Test cases and Results

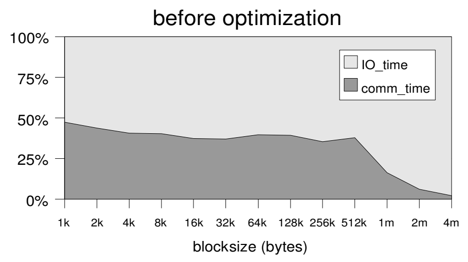
We use two parallel I/O benchmarks (FLASH I/O and IOR) and one real scientific application (CHIMERA) to test Lustre ADIO driver. And all the tests were performed on ORNL Jaguar supercomputer.

3.1. IOR benchmark performance

We simulate the collecting I/O behavior in IOR source codes instead of Lustre ADIO driver directly because we can't access and modify Cray MPI lib on Jaguar.

We compared Lustre ADIO driver(collective/independent) and POSIX on Jaguar in IOR benchmark with 1024 clients and 72 OSTs. Stripe size was 4MB and blocksize was set from 1KB to 4MB exponentially. IOR command is: "IOR -a API -b blocksize -t blocksize -w (-c) -v -o outfile -q".

To analyze the communication overhead, we separated communication from I/O operation. That means for each collection, information communication completes first and then I/O write. The comparison analysis in Figure 4 shows that with our optimization for MPI collective communication, the communication overhead was reduced from 50% to 20% significantly.



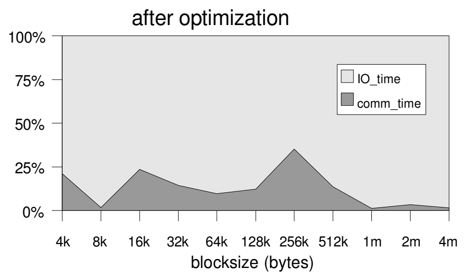
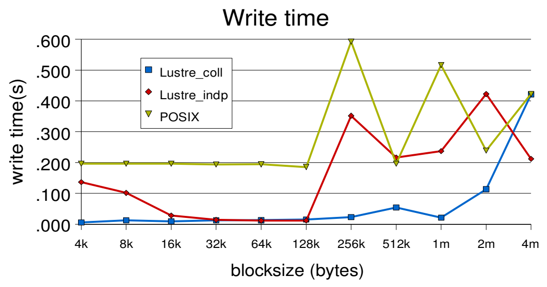


Figure 4. write time overhead analysis on Jaguar

The write performance in Figure 5 proves that after simplifying MPI communication, the write time of collective I/O was much lower than those of independent I/O mode and POSIX. But when block size became larger, the write time increased too. So, as mentioned in application access pattern identification, for big size request collective write driver won't help much because this kind of I/O pattern already can get good performance itself. But, although write time of collective IO was very small, collective open time took much, as shown in Figure 5. The collective open time was about 30 times of write time and it had a great impact on the collective IO performance.



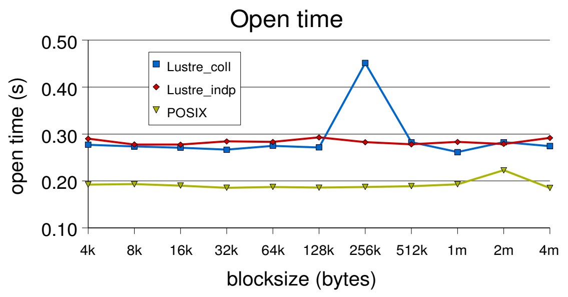


Figure 5. write time and open time on Jaguar

3.2. FLASH I/O benchmark performance

FLASH I/O benchmark[5] is a popular HPC scientific application benchmark. It measures the performance of the FLASH parallel HDF5 output. It recreates the primary data structures in FLASH and produces a checkpoint file, a plotfile with centered data, and a plotfile with corner data. The plotfiles have single precision data. The purpose of this routine is to tune the I/O performance in a controlled environment. The I/O routines are identical to the routines used by FLASH, so any performance improvements made to the benchmark program will be shared by FLASH.

In FLASH I/O benchmark, each client could have different block size. By default, each 3 clients write 320k, 324k and 328k data respectively, which would cause extent lock conflicts. To observe the behavior of collecting small I/O requests, we shrink block size from $8 \times 8 \times 8$ to $4 \times 4 \times 4$ and enlarge the iteration by increasing the number of the variables from 24 to 1024 per array element. In our test, each 3 processes should write 40k, 40.25k and 40.5k data respectively. In this I/O pattern, each about 26 processes send their data to one I/O client. The number of the processes scaled from 4 to 1024 exponentially. We set stripe_size=1MB and CO=4, which means each OST will be accessed by 4 I/O clients. Also, POSIX was used for performance comparison.

The results in Figure 6 show that for the different system scale, the output runtime of our driver was always half of the one of POSIX, except that when nprocs=128. Since the number of blocks is fixed in each process, as we increase the number of MPI processes, the aggregated I/O amount linearly increases as well. When nprocs=512, Lustre ADIO collective write driver achieved its best write bandwidth of 624MB/s, which was much higher than 368MB/s POSIX done.

# of procs	Lustre_coll(s)	POSIX(s)	File size(MB)
4	2.03	9.87	160.75
8	2.74	10.3	321.5
16	4.21	10.78	643
32	5.32	12.26	1286
64	6.6	13.04	2572
128	20.3	23.24	5144
256	23.55	36.07	10288
512	30.07	55.88	20576
1024	65.88	106.86	41152

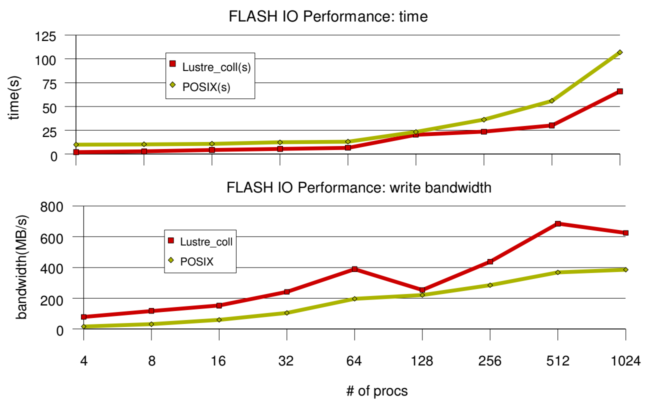


Figure 6. FLASH I/O performance on Jaguar

3.3. CHIMERA

CHIMERA[5] is a multi-dimensional radiation hydrodynamics code designed to study core-collapse supernovae, one of the major applications in ORNL. The code is made up of three essentially independent parts: hydrodynamics, nuclear burning, and a neutrino transport solver combined within an operator-split approach. The multi-physics nature of the problem, and the specific implementation of that physics in CHIMERA, provide a rather straight-forward path to effective use of multi-core platforms in the near future.

We ran CHIMERA with 512 processes and 1MB stripe size. In our test, CHIMERA output 8 restart files in HDF5 format. Figure 7 describes the I/O pattern conversion. Obviously, the original lots of small size I/O requests were merged into the fewer bigger size I/O by Lustre ADIO collective write driver.

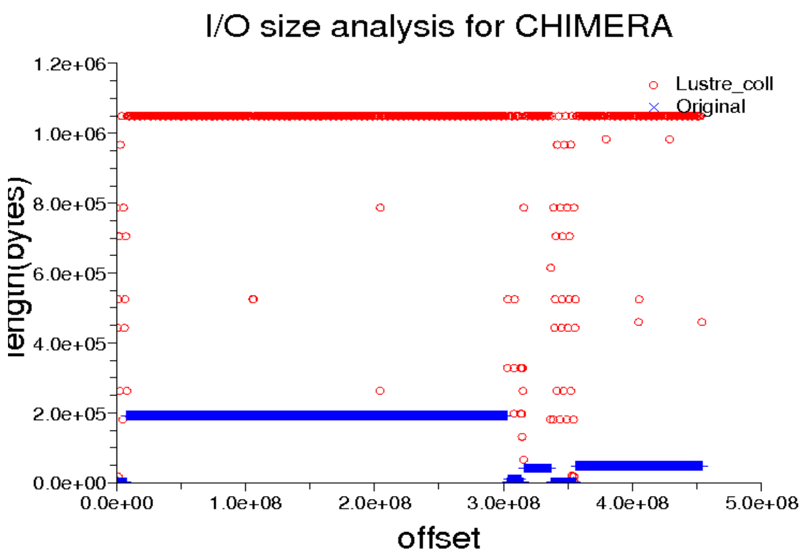


Figure 7. CHIMERA performance on Jaguar

Table 1 gave the output time of 8 restart files. Item 2 showed a 100x speedup and the overall did a 10x speedup from original 362(s) to the current 32(s). It proves the performance was improved significantly.

Table 1. The output time of all CHIMERA restart files

Restart File No.	Lustre_coll(s)	Original(s)
1	7.431355953216553	16.74987316131592
2	2.910053014755249	224.6195049285889
3	2.853060960769653	15.89030098915100
4	3.058269977569580	24.44141197204590
5	4.030251979827881	15.76727080345154
6	3.008349895477295	17.22050786018372
7	2.960572004318237	21.45163202285767
8	5.498844146728516	25.52246379852295
total	**31.750757932663**	**361.662965536118**

4. Conclusions & Future work

In this report, implementation and performance evaluation of Lustre ADIO collective write driver is introduced. Lustre ADIO collective write driver can collect and reorganize data to produce the pattern(stripe-contiguous) Lustre prefers with low overhead. To improve it, we provide several MPI hints to check application I/O pattern, keep load balancing between clients and OSTs, and reduce communication overhead, effectively. The benchmark results show that Lustre ADIO collective write driver can achieve much better parallel I/O performance independent I/O and POSIX with proper stripe size and hints setting, especially for small size I/O. However, for other ones, due to high collective open time and inevitable barrier operations (MPI_Waitall) between each data collection, its performance still needs to improve. At last, the CHIMERA results prove Lustre ADIO collective write driver did improve significant performance for some real scientific applications.

At present we only focus on the collective write, in the future we will investigate collective read, and some I/O optimization features[6] will also be exported to Lustre. We hope Lustre ADIO driver can contribute to more scientific applications.

5. References

[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] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, "MPI: The complete reference". The MIT Press

[4] FLASH IO Benchmark:http://www.ucolick.org/~zingale/flash_benchmark_io/

[5] Supernova simulation with CHIMERA,

http://nccs.gov/wp-content/uploads/2007/08/messer_paper_thursday.pdf

[6] "HLD of exporting I/O features" by WangDi

Appendix

We add several hints to Lustre ADIO collective write driver for performance tuning. They are listed as follows:

- **directIO**
  Enable/disable direct IO by setting "enable"/"disable"
- **CO**
  In stripe-contiguous IO pattern, each OST will be accessed by a group of IO clients. CO stands for **C**lient **O**ST ratio, the max. number of IO clients for each group.
- **ds_in_coll**
  Collective IO will apply read-modify-write to deal with non-contiguous data by default. However, it will introduce more overhead(IO operation and locking). To avoid this, we can use the hint to disable this processing.
  Here, we use ds(data sieving) instead of read-modify-write in the hint name because data sieving is used more commonly than read-modify-write.
- **big_req_size**
  During IO pattern identification, we won't collect the data if its size is big.
  We give this hint to define big request size.
  Usually, for Lustre, when request size is no larger than 4k or 8k, the performance of collective IO will be better than POSIX. Of course, the users can define it as they like.
- **same_io_size and contiguous_data**
  These are two hints to tell the driver whether each request IO size is same and whether the request data is contiguous. If they are both "yes", we can optimize ADIOI_LUSTRE_Calc_others_req() by removing MPI_Alltoall(), because each process can easily calculate the pairs of offset and length according to continuous and same size data without collective communication.
  What’s more, currently only when they are both positive, the optimization can work. In the near future, maybe some efforts will be made to other conditions.

You can use these hints in IOR hint file by option “-U”. For example,

```
IOR_HINT__MPI__directIO=disable
IOR_HINT__MPI__CO=1
IOR_HINT__MPI__same_io_size=no
IOR_HINT__MPI__contiguous_data=yes
IOR_HINT__MPI__ds_in_coll=disable
IOR_HINT__MPI__big_req_size=40960
```