Report of test on PLFS with MPI/IO API and some possible improvements*

*Note: This is a experinmental Test.

1st Yiwei Victor Yang
Schools of Information Science and Technology
ShanghaiTech University
Shanghai, China
yangyw@shanghaitech.edu.cn

***Abstract*—The Big Data today requires larger and more complex parallel database to store more information. The work before has a good performance in N-N I/O, but find N/1 not as Ill as it. This work is a implementation work of MPI IO test of PLFS, a distributed argument chunking layer that can be deployed on modern Distributed File System(DFS) like GFS, GPFS, HDFS, Spark and other DFS. For use of alog-structured file system, where write operations are performed sequentially to the disk regardless of intended file offsets and file partitioning, where a write to a single file is instead transparently transposed intoa write to many files, increasing the number of available filestreams, PLFS has been the good choice for PFS.**

***Index Terms*—File System, N-1, Parallel, I/O Performance**

I. Introduction

Much of the high performance computing (HPC) industry has focused on the development of methods to improve compute-processing speeds, like Infiniband by Mellonax and NVMe. This kind of speed up creates a tendency to measure intercommunication performance.

For large and complex parallel nowadays, they contains a huge amount of components that must protect them from interrupts caused by component failures. If I can let the File System know the log and recordingly do the checkpointing of the state by the semantics of the log, the persistent storage can be done with loIr cost. Typically HPC applications protect themselves from fail-ure by periodically pausing and concurrently writing theirstate to a checkpoint file in a POSIX-based PFS.

As for different checkpoint patern, I have 3 different ones for N-N, N-1 segmented and N-1 strided. The pattern for the parallel application is same consiting of 6 preocesses and spread across 3 nodes.

The more naive implementation with less metadata-intensive I/O pattern may be N-1:

I specify the N-1 as N-1 strided rather than N-1 segmented in the rest of the paper. In this mode, processes write to multiple small regions at many different offsets in the file. The challenge with PFS is that these offsets typically do not coincide with file system block boundaries, which results in multiple concurrent accesses to the same file at different locations. As a result, the HDD (hard disk drive) must perform many seek operations and multiple writes to the same file system block will be serialized. As a result, current PFSs serve N-1 very poorly, especially at very large scales. To a lesser extent, PFSs must implement a file-area locking scheme to ensure POSIX compliance.

Identify applicable funding agency here. If none, delete this.

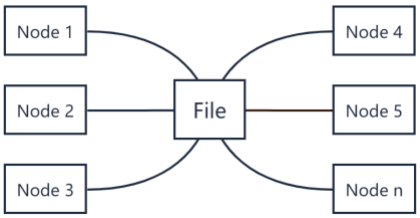

Fig. 1. N-1 I/O pattern

The performance issues with N-1 is somewhat overcome its simplicity, N-N becomes the mainstream and can obtain a good performace on GVFS or even Spark as mentioned in their survey thesis. The main difference betIen the N-1 and N-N is in N-N, each process does I/O to handful of reasonabl=y sized files. Fig.2 depicts the I/O pattern where File can be more than one file, but usaully just a few.

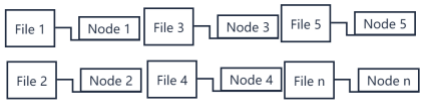

Fig. 2. N-N I/O pattern

Although the N-N modell is showing some disadvantage in scaling files in a single directory (multiple files adjacent in memory), PLFS distributes the files to multipule directories to alleviate overloading one directory with too many files.

The reason why many user prefers N-1 checkpoints (BTIO, IO 500, FLASH IO and many other io benchmarks utilize it) is

they prefer to manage 1 file rather than thousands of them. The N-N pattern cannot avoid mapping because of N-M restarts.

N-N writing is easier for lock-happy file systems, which means it suffer lock convoys and other congestion when many other different clients are concurrently writing data in the same region. The rest of the papers is organized as follows. I present more detailed background and motivation discussed in the original paper and provide some recent view on it in Section II, describe my method for evaluation process and result by MPI I/O method and gdb debug with my configuration in Section III. I preset the possible improvements in the Section IV. The Last Section will be the summary I learned in the whole process.

II. BACKGROUND

A. PLFS

PLFS is an interposing filesystem that sits between HPC applications and one or more backing filesystems. They both sit on the OST node, and can be accessed by the Computing Node by loading PLFS MetaData like reading Inode in the normal linux system. PLFS has no persistent storage itself. PLFS trans-parently translates application-level IO access pattern sothat they perform well with modern DFSs. Neither HPC applications nor the backing filesystems need to be aware of or modified for PLFS to be used.

B. PLFS Structure

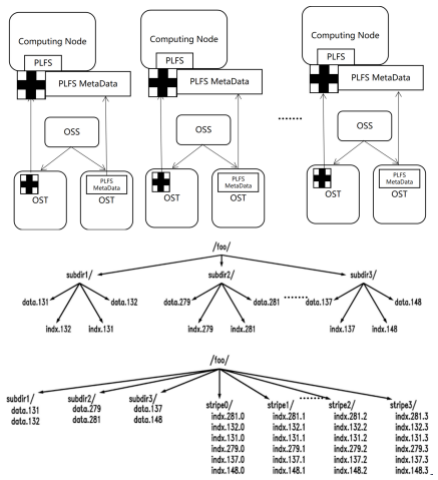

Fig. 3. PLFS general graph (OST means object storage targets)

In the Fig. 3, we can clearly see that each file stores in the PLFS is mapped into one or more container directories. Fig. 4, for example, shows a N-1 checkpoint file called ckpt being written into a PLFS file by 3 processes distributed across computing nodes. The PLFS virtual layer shown in the above Fig.3 is backed by n volumes of a DFS. Each process' blokcs of data written into the checkpoint file are strided, resulting in a process' blocks being logically distributed. The logical view and the physical view of PLFS' difference lies in it throughout the checkpoint file.

C. PLFS log-structured Metadata

When every process writes to the shared logical file, PLFS appendsthat data to a unique physical logfile (data dropping) for that pro-cess and creates an index entry in a unique physical index file (indexdropping) which maintains a mapping betIen the bytes within thelogical file and their physical location within the data droppings.When a read request (e.g.read(fd,off,len)) is performed, PLFSqueries the index to find where that actual data resides within thedata dropping files. The key variables of a current index entry are:

- *logical offset* where the data is, from the application's per-spective in a single logical file
- *length* number of bytes written
- *physical offset* this is the physical offset within a contiguousd ata dropping file
- *chunk id* the ID of the dropping file where the data resides.

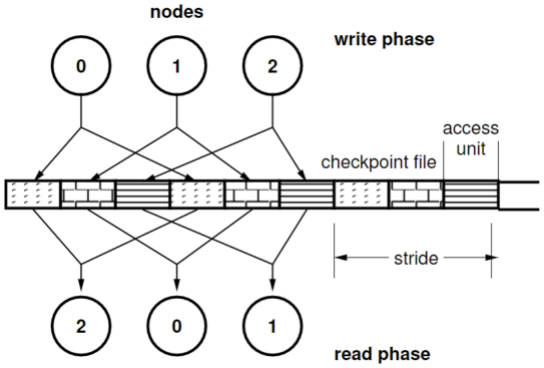

Fig. 4. Checkpoint benchmark operation

D. PLFS extensibility

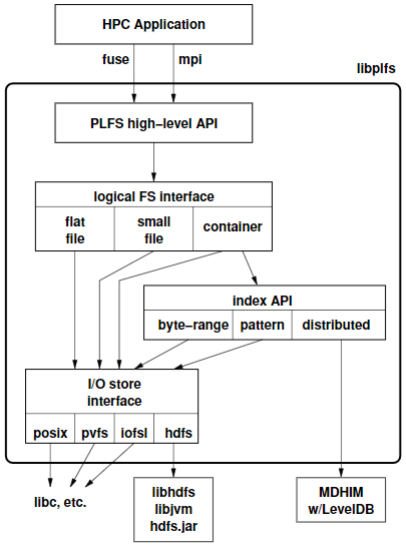

Fig. 5. PLFS extensibility

There are basically three interfaces to use PLFS. They first wrap a Abstract Devise Interface for I/O(ADIO) which has some introduction in *mpi* derectory. In my evaluation, I don't need to apply the patches for openmpi to do MPIIO test for the fs_test just do the FUSE API. The second is File System in User Space (FUSE). This is a FS done by writing files in the user space and communicate with the FS. They also provide their own PLFS-specific API for linking usage. It does not require additional System Administration work to provide PLFS FUSE mount points. The test carried in the orinal paper shows the PLFS API yield good performance.

Besides thanks to PLFS high-level API, PLFS Logical Filesystems is allowed to use non-mounted non-POSIX filesystems to store application data.

III. Evaluations

A. MPI I/O Method

MPI for Parallel I/O is a parallel I/O system for distributed memory architectures will need a mechanism to specify collective operations and specify noncontiguous data layout in memory and file. It will chunk the reading data into pieces and creating MPI process to paralelly read or write, which is like receivinig and sending messages. Hence, an MPI-like machinery is a good setting for Parallel IO (think MPI communicators and MPI datatypes). Specially in file accessing, they utilize individual file by function *MPI_File_seekMPI_File_read* and calculate the byte offsets in the *MPI_File_read_at*. The MPI I/O API is designed over FUSE which the MVP code for testing is listed as follows:

```
MPI_File fh;
for (i=0; i<BUFSIZE; i++) buf[i] = myrank *
    BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD,
    "testfile",MPI_MODE_CREATE |
    MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, myrank * BUFSIZE *
    sizeof(int), MPI_INT, MPI_INT, "native",
    MPI_INFO_NULL);
MPI_File_write(fh, buf, BUFSIZE, MPI_INT,
    MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

1) Environment Configuration: I run all the data with openmpi 3.1.6 on wsl2 with kernel version 4.19.121. The filesystem is ext4 and the data is stored on HDD. The fuse runs on the internel File System on Distro Ubuntu 20.04. GCC version is 4.8.5. GCC version 7 may come into error for *operator ¡¡* and *no type named 'type'* The version control tool is spack.

2) Configuration: The config file in the *${HOME}/.plfsrc*. I specify the mounting source in *mntplfs* and the destination in /root/CS130P/mount. I made a direct I/O comparison in /root/CS130P/compare.

B. Compilation and test

The code can be compiled with the gcc-4.8.4 specified. *cmake -DCMAKE_CXX_COMPILER=g++48 . && make && make install*. After installation, the plfs is default installed into */usr/local/lib*. I switch the compile option into $-g$ for debugging. Then I mount the file by simply *plfs /root/CS130P/mount*. In that derectory, I wrote a simple code to generate files using FUSE API on them and get the tracing result to the diagram in the Appendix.

For MPI I/O, after installing the openmpi using spack and compile it, I can run command like *mpirun -np 4 ./test.x -type 2 -strided 1 -op write -nobj 1024 -size 1048576 -chkdata 0 -sync 1 -target ../mount/test-file*. The different process number can be changed by the parameter of $-np$. I both tested the read and write FUSE API with and without PLFS. After multiple test changing the variables of *nobj* and *size*, I found them overall has a descending trend. Here's one of the plot of the raw testing using *-size 1024 -nobj 1048576*. Every time when running, I make sure that it's not running multiple same processes rather than chunked file for different ones by using strace.

C. Result

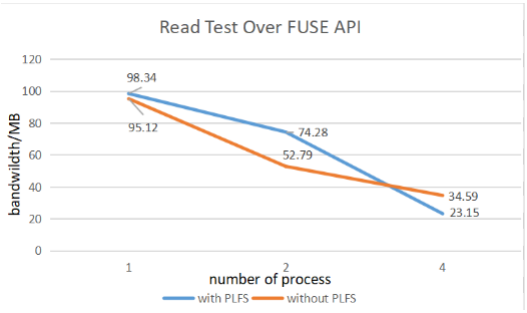

Fig. 6. MPI I/O test reasult

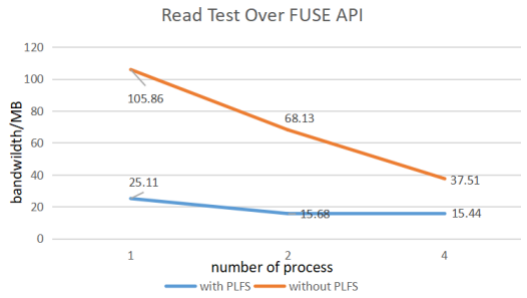

Fig. 7. MPI I/O test reasult

a) Why more threads do not generate better result for PLFS?: Because although the PLFS is sure to have a performance jump in DFS, but on FS like ext4, POSIX API is already take full use of the single disk. Adding more threads will not speed up the I/O or may generate some lay backs.

b) Why PLFS is slower than the direct I/O?: If the filesystem is not on the DFS and do not have multiple OTS connected by high bandwidth switch. The overhead of creating log structure striding may definitely down performance.

IV. Possible Improvements of PLFS

a) Compression of the Metadata: The log file is maintained in the struct in $plfs\_private.h$, which can be a huge space overhead. I can simply store them as compressed file or serialized them to seek the balance betIen the space and read speed.

b) Take use of more information by pattern detection: Just reading log information in PLFS is not as efficient as reading the semantics of the log and predict next operation. If I come up with a heuristic algorithm that is aware of the previous operation and prefetch the next requested data, no matter read or write will be very helpful. The prediction can be done by the Markov Process.

Summary

A. Practical Skills

a) GDB debugging skills: For the dipiction of diagram, I utilize gdb to find which function PLFS use which is a better way than static analysis.

b) unit test writing skills: When I was specifying MPI I/O parameter size as 1, the PLFS mount point crashes. I wrote some test cases only take care of read of random int32. I got into the bugs by knowing that *plfs_read()* does not support size of 0 file and that was not error handled by the PLFS.

B. About how to back testing a academic paper

The process of knowing one field is to first read the survey paper of that field and PLFS is definitely a good start up paper for File System learning in HPC. The steps in the tutorial let me know how to maintain a project served for the paper. PLFS actually provide some thoughts for the future design of the PFS and feed other improvements of it.

Acknowledgment

The author is thankful to Prof. Shu Yin and all the teaching stuff of the Operating System Project (CS130P) ShanghaiTech for their providing such a good material to research on. Also, the author is grateful for those, especially fellows from LANL who has devoted their career in the Parallel File System. I knew it's very hard to debug and make applicable for any ideas by compiling and benchmarking the PLFS. Their efforts push the fields of HPC forward.

References

[1] Zaharia, Matei, et al. "Fast and interactive analytics over Hadoop data with Spark." Usenix Login 37.4 (2012): 45-51.

[2] Reyes-Ortiz, Jorge Luis, Luca Oneto, and Davide Anguita. "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf." INNS Conference on Big Data. Vol. 8. 2015.

[3] Heichler, Jan. "An introduction to BeeGFS." (2014).

[4] Bent, John, et al. "PLFS: a checkpoint filesystem for parallel applications." Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. IEEE, 2009.

[5] Mehta, Kshitij, et al. "A plugin for hdf5 using plfs for improved i/o performance and semantic analysis." 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. IEEE, 2012.

[6] Cranor, Chuck, Milo Polte, and Garth Gibson. "Structuring PLFS for extensibility." Proceedings of the 8th Parallel Data Storage Workshop. 2013.