Project Report and Review on Parallel and Distributed File Systems

Zhijie Yang
School of Information Science and Technology
ShanghaiTech University
Shanghai, China
yangzhj@shanghaitech.edu.cn

***Abstract*—This is a literature review on parallel and distributed file systems as the project of CS130P. In this review, the background knowledge of parallel and distributed file systems has been introduced. Three representative distributed file systems, among which two of them are parallel file systems, are mentioned, concluded and analyzed. These three different file systems are also been compared together to give a more clear understand of their characteristics.**

***Index Terms*—parallel file system, distributed file system**

I. Introduction and Background

According to the hierarchy of memory in a computer, the block device, i.e. the disks are usually the slowest devices both in terms of throughput and latency but with relatively the largest capacity. The most commonly used approach for a single-node computer is to implement a redundant array of independent disks (RAID). With help of RAID, the throughput and stability of disks has been greatly improved even through these disks are running under an unscalable file system. However, this only works fine under very narrow conditions — there can only be one node, which is the computer itself in this case, accessing this array. This of course will not cause any problem if there is only one computing node. Alas, the trend of high performance computing is clustering, which aggregates multiple computing nodes together to fully parallelize the tasks. As the computing power is no longer a bottleneck, the I/O performance, especially the scalability now becomes a new critical problem.

To cope with, distributed file systems has emerged. They usually assume different computer nodes are interconnected with high speed local area network (LAN) or special channels like Myrinet or InfiniBand. With these fast interconnections, accessing the disk drivers from other computing nodes no longer greatly suffers from throughput limitation in the interconnection. Such kind of network dedicated for storage is called storage area network (SAN). To utilize this interconnection to reach high throughput, a major issue for a distribute file system is to distribute directories, files and even different chunks of a file into different disks. The grain of such distribution in some primitive implementations are coarse, for instance, they just simply partition different files in a directory into different disks, which can not necessarily increase the parallelism of data. Besides, some distributed file systems require central servers, where the scheduling and lookup requests are handled. These designs are basic and lacks both scalability and limits the performance gain. If these problems can be handled, most of the overheads in a distributed file system can be reduced or even cancelled.

If a distributed file system supports parallel applications, it is called a parallel file system. This kind of file system may support simultaneous reads and writes from or to a same file by multiple processes. It focuses on providing extreme throughput while can still guarantee the correctness of data.

As there are different type of such file system, I picked some of the representative works, which is GPFS by IBM [2], PVFS by Clemson University and Argonne National Laborartory [1] and zFS from Haifa University [3].

This review is constructed as follows: In section 2, a short summary on GPFS is conducted. In section 3, zFS is concluded and PVFS is reviewed in section 4. Some comparison is given towards these three file systems in different aspects in section 5.

II. GPFS

A. General introduction of GPFS

General files systems running on clusters leverages logical volume manager (LVM), which may have addressability limitations on logical volumes. GPFS takes another solution: it uses striping in the file system. It is capable of handling 4096 disks with each disk capacity no greater than 1TB. It also support a single file to be as large as $2^{63} - 1$ bytes. To exploit parallel reading mechanism, it partitions large files into blocks and put consecutive blocks on different disks for greater throughput. It also implemented sub-blocks that reduces internal fragments on storing small files when using large block sizes. GPFS also take the fashion of write-back buffer cache with prefetching. When the sizes of the disks in this system differs, GPFS asks user to make a trade-off between space utilization and throughput. GPFS uses extensible hashing to handle directories with great amount of files (or sub-directories). It also implemented logging on metadata updates during writing. Once a writing is done, the logged metadata is relinquished. These basic attributes of GPFS promises its scalability as well as its potential improvements resulted from optimizations.

B. Parallel I/O

The design philosophy of GPFS is using distributed locking as the base and take centralized managements as an higher level commander. GPFS leverages byte-range locking to write data. As these locks can only be effective for one byte as minimum and the whole chunk of blocks of the file at maximum, the parallelism can be assured (except for some cases will be discussed later). Centralized hints for disk space allocation is also available in GPFS. The allocation map is also specially prepared for better parallelism. The map is divided into n lockable regions containing every disk's $1/n$-th allocation status, such that different nodes can directly allocated spaces on their own, without lock conflicts. They then just need to report their allocation map usages periodically to the allocation manager and only when a node runs out of free disk spaces in its region will it request for a new region. Token is also introduced into GPFS. When multiple nodes want to write to the same block, a token conflict will occur. On this occasion, GPFS allows disabling POSIX semantic to forward the read and write request to the only one node responsible for a specific data block. This locking hierarchy provides GPFS with decent parallelism in the lower level and also the easiness of implementing centralized scheduling of locks and tokens.

C. Fault Tolerance

Since GPFS is a logged file system, with the restrictions of its distributed locking protocol, when a node fails during writing procedures, the metadata changelog , the cache to write and the locks for updating blocks can be easily accessed by other nodes according to the command of GPFS. Once the write is completed by other nodes, the locks held by the failed node is released such that all the other nodes will be able to take over this failed node's role. GPFS allows nodes to work independently in case of communication failures, and disk fencing is used to resolve a communication failure under a two-node configuration. To handle disk failures, the first and simplest measure to take is enabling RAID. RAID allows the system to make out the failure of a disk. In addition to RAID, GPFS also have its method — replication. GPFS can write data to two disks in same locations at the same time, and take one disk as the backup of another when there is a failure. Interestingly as the authors mentioned in the conclusion of the origin paper, data replication is just a solution to avoid the expensive RAID system. However, as the price of RAID has come down afterwards and the development of new RAID solutions as well, modern mainstream systems are shifting to pure RAID schemes instead of using replication provided by GPFS (actually only slight difference with RAID 1).

D. Scalability of Online Utilities

GPFS allows modifying the configuration of a system by add, removal and replacement of disks. Before removal or replacement, the data in affected disks must be moved. This requires traverse along all the metadata and indirect blocks in every disks, which is costly in time. GPFS select one node to be the file system manager which assigns a range of inodes to other nodes and let them process these requests in a specific range of blocks. GPFS also allows these utilities run online — without letting these part of data being inaccessible during process. However, special locks may be required when processing over higher-level metadata. With online utilities available, users no longer need to worry too much about the suspend of the system before making any hardware modifications to the file system. This will benefits especially commercial users in the availability of the whole cluser computer.

E. Summary

GPFS as the first shared file system of IBM, has great compatibility since its POSIX-identical semantic. Plus its simple but efficient low-level abstraction and management strategies, GPFS is of great potential in throughput, enabling the performance increments in cluster computers.

III. PVFS

PVFS as a parallel file system with novel design and implementation, have the following objectives:

- Uncentralized.
- Robustness and scalability.
- Distributed data and metadata.
- High throughput.
- Easy of use.
- Support for multiple APIs.

A. Design Philosophy of PVFS

PVFS is designed to be mainly on user-level. In most cases, it does not require users to have their kernel modified. Instead, it is based on existing local file systems. PVFS does not require other message passing mechanisms aside for simple TCP protocal.

1) Metadata Management: PVFS strips the payload of files accorss the I/O nodes. It records how a file is stripped, i.e. into how many nodes, which is the first node, how large is the strip size in the file's metadata. All the access to these metadata is managed by the manager daemon in PVFS.

2) Data Storage: In PVFS, consecutive stripes of a file are mapped into different I/O nodes to ensure the maximum data parallelism. When a user application is accessing a file stored in PVFS, it will connect with those I/O daemons (which runs on different I/O nodes) informed by the PVFS manager. Determining the I/O stream for an access is simple in PVFS — combining the pieces of intersection of the available physical stripe on the specific I/O daemon and the requested logical partition by the application together. This approach is simple and efficiency.

3) Application Programming Interface: This is the most important part of PVFS for its top-level compatibility. It can trap Unix I/O system calls. It changes the system call wrapper during linking procedure to make it possible for a normal system call to call into the functions implemented by PVFS when this access is towards a PVFS file. This method is not able to handle the child process's behavior. An alternative is

to mount this file system to the operating system using hooks provided by Linux for new file systems. This enables users add PVFS to their cluster without modifying any part of their code or to recompile the Linux kernel. Another API is via MPI-IO interface via ROMIO implementation. Contigious file access can be greatly optimized with this interface. The third API is a set of native API provided by PVFS. This provides better performance since all the access is relatively direct (require fewer system calls) compared to the other two method, but it requires the programmer to change normal system calls into PVFS native APIs in their code and a recompile is necessary.

IV. zFS

A. Introduction

zSF is a decentralized file system, which performs over a set of high-speed network connected computer cluster. zFS is designed to be scalable for a cluster with number of computers ranging from only several to thousands of computers. To earn availability for general users, zFS was not made to be hardware-specific — normal commodity components are sufficient to compose a storage system taking advantages from zFS.

In this review, we will overlook zFS from its functionality parts and its architecture. We will also analyze it together with other existing cluster file systems to demonstrate its uniqueness and usefulness.

B. The Design of zFS

1) Design Goals: The design concepts of zFS is worthy of introduction. Its uniqueness all come from these concepts:

- Based on object store devices (OSDs).
- Almost linear scalability for no matter few or many machines.
- Abstraction of global cache from the aggregation of all computer memories.

2) Functionality Parts: The implementation and decision of functionality parts both follow the concepts above.

- Object Store: zFS is based on OSDs. Object store has been verified to be efficient for clusters. However, to make things tidy in zFS, the team chose to build zFS totally based on OSDs, which has most functions handled in the store device instead of the file system, and thus makes object store transparent.
- File System Layout: In zFS, both the files and the directories are considered as a file (similar for a inode base file system). There exists a one-to-one image from a directory or a file to a single object stored in this file system. By using object stores, many higher level functions and services provided by OSDs are now available to users.
- Front End: The front end (**FE**) of zFS runs on every client node of a cluster that uses zFS. It is designed to provide file system interfaces to the client.
- Lease Manager: Leases are used in zFS to prevent a dead lock when a computer is not able to respond (e.g. broken connection or a computer failure). Leases are locks with expiration time. When the holder of a lock is unable to respond, just simply wait for the lease to expire. Leases are managed by lease manager (**LMGR**), which is mainly responsible for acquiring the major-lease and guarantee the mutual exclusion access for objects on the OSDs.
- File Manager: File managers (**FMGRs**) is used to manage opened files. **FMGR** is file-specific — each opened file has its own file manager. It exists to mediate all the lease requests for any access requests to the file. If the file to be opened locates in other computer's cache, the **FMGR** will forward the data from the other computer's cache to the local cache.
- Cooperative Cache: Since the network is fast enough, it is reasonable that fetching a data block from other computer's cache is faster than reading it from a local disk (by data hierarchy). Therefore, The cooperative cache that aggregates the cache in all the computers may ensure the almost linear scalability.
- Transaction Server: The operations associated with directories in zFS could be complicated since modifying a file in a directory requires both the modification for the directory object but also the file object. Transaction server (**TSVR**) is introduce to mediate these operations.

3) Protocals: Due to the lenghiness, protocols in zFS will not be discussed in detail here. However, its core is cascading request and responds to ensure data locality — **FE** send requests to **FMGR** (or to the **TSVR** first and then let **TSVR** send to **FMGR** if it is an operation associated with directory), **FMGR** sends requests to **LMGR**, **LMGR** accesses the **OSD** (on normal read or write), or lets the **FE** directly access **OSD** (on flushing and cache-to-cache read).

C. Fault Tolerance

FE failures are the simplest. Just waiting for the lease of the failing **FMGR** to expire and the control is regained. Limitations on number of dirty pages in local cache to the amount of pages can be safely written to OSB in time far less than lease renew time may resolve **FMGR** failure. On **LMGR** failure, **FMGR** singals all the **FEs** holding the leases to flush the data to **OSD**s. **OSD** failure is the hardest, it cannot be handled without replica of data.

D. Comparison with Other File Systems

In this subsection, we will compare zFS against two common and well-known file systems — Luster and GPFS.

1) Luster: Luster provides clients a standard POSIX interface, and the managements are mainly done in its cluster control systems. It assumes OSDs are programmable, which is called storage targets in Luster. zFS does not assume OSDs to be programmable, but requires the OSDs to provide a non-standard locking interface.

2) GPFS: le GPFS is developed by IBM as a file system for high performance computing clusters. It is based on standard disks and achieve failure tolerance by consensus style solutions while zFS uses OSD can treats failures with OSD base schemes.

E. Conclusion

This work presented a novel OSD based file system with high scalability. Based on OSDs, this file system has reached tidiness in its implementation and also evaded complicated implementations of object store in normal disks. In addition to existing file systems, zFS has its advantage of open-source and referring to a broad series of research result, it has many improvements over other file systems. It is possible to predict that zFS will mature and become a popular file system for ubiquitous clusters.

V. Comparison

Among these distributed file systems, GPFS is the most mature one, which is not free of use indeed. GPFS build the whole file system from the very bottom layer — communication with block devices, bytewise lock, to the higher level utilities — high scalability and the ability of adding,[1] removing or replacing of a disk without suspending the whole[2] cluster. zFS, took yet another totally different approach. It uses OSD as it basic device. Similarly to GPFS, zFS also started from the very beginning of a file system. However, since it uses OSD, some implementations is able to kept[3] tidy. Compared to GPFS, zFS is just like a beginner but it is aimed for those requires less have write-sharing. zFS[4] may still perform close to GPFS in less sharing situations according to its developers. PVFS has the most differences with the other two file systems. Firstly it almost have no[5] implementation for direct access to block devices. It is a virtual[6] file system (based on existing local file systems). It separates[7] the metadata and the file data in some ways, this resembles[8] object storage to some extent but the file systems it uses are all based on normal disks. PVFS is said to have optimization for throughput, which is true at least for local test, but it also consumes a great amount of CPU resources (taking up more than 100% percentage of CPU usage on an i7-9750H CPU, giving around 1200MB/s throughput with both server and client on the same aforementioned machine).

VI. Conclusion

Parallel and distribution file systems have been developing for more than 20 years. These work aforementioned is just three of the very beginning works (GPFS is in 2002, PVFS is in 2000 and zFS is in 2003). They provided the academia with different methodologies and views towards distributed file systems. Although some of them is not mature enough for business and everyday usage, their engineering inspirations have been adapted by the successors.

Installation Report

The installation procedures is quite the same as building other software from source. It basically follows the following steps:

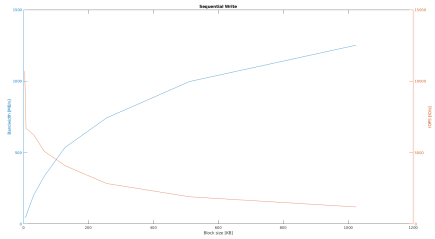
- Configuration for compilation
- Compilation
- Copying binaries and setting links
- Generation of configuration files

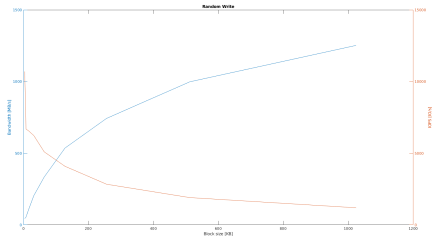
Since my computers are all equipped with solid state drivers (SSDs), whose consecutive I/O throughput can reach more than 4000Mbps. The throughput of block device is no longer a bottleneck. Instead, the interconnection between the client and the server will be limiting the throughput due to the commodity 1000Mbps cabled network interface controller (NIC), whose maximum theoretical throughput is 5 times slower than the block device. To exclude the potential throughput limitation introduced by network infrastructure, I decided to configure both the server and the client onto one local machine with an SSD connected using NVM Express (NVMe). The performance of OrangeFS can be seen in the following section.

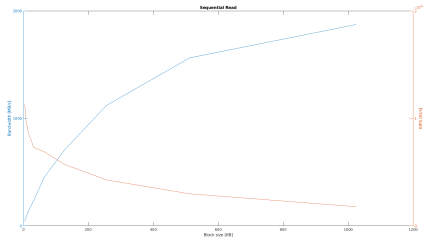
For simplicity, I put all the commands needed to run orangefs into a shell script according to the configuration mentioned above.

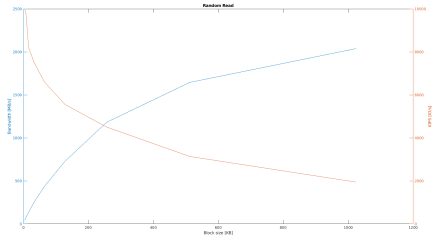
```
#!/bin/sh
sudo /opt/orangefs/sbin/pvfs2-server -a
    localhost /opt/orangefs/etc/orangefs-
    server.conf # Starts the server
    service on localhost
sudo modprobe orangefs # Inserts the
    kernel module
sudo /opt/orangefs/sbin/pvfs2-client -p /
    opt/orangefs/sbin/pvfs2-client-core #
    Starts the client service on localhost
if [ ! -d "/mnt/orangefs" ];then
sudo mkdir /mnt/orangefs
fi
sudo mount -t pvfs2 tcp://localhost:3334/
    orangefs /mnt/orangefs # Mounts the fs
     to /mnt/orangefs
```

Performance Evaluation









SUMMARY FOR OPERATING SYSTEM COURSE

I personally learnt a lot via the hands on practice of the basic concepts of operating systems. This course greatly helped me to have a deeper understand of how a computer really works. Although some of the projects are a little bit too hardcore, but the feeling of passing all the test cases is really good. I personally have the poorest score in the project of file system but I think I can still have a better understand of it via this project.

REFERENCES

[1] Philip H. Carns, Walter B. Ligon III, Robert B. Ross and Rajeev Thakur. 2000. PVFS: A Parallel File System for Linux Clusters. In Proceedings of the Extreme Track: 4th Linux Showcase and Conference, October 2000.

[2] Frank Schmuck and Roger Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)

[3] O. Rodeh and A. Teperman, "zFS - a scalable distributed file system using object disks," 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings., San Diego, CA, USA, 2003, pp. 207-218.