2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems

Optimization Of Ceph reads/writes based on multi-threaded algorithms

Ke Zhan*†, AiHua Piao*†
*Institute of Information Engineering, Chinese Academy of Sciences, Key Laboratory Of Network Assessment Technology

†Beijing Key Laboratory Of Network Security And Protection Technology

Beijing 100093, China
Email: zhanke@iie.ac.cn, piaoaihua@iie.ac.cn

***Abstract*—Ceph is an open source distributed file system. Based on two methods of command line and library librados, we implement the files reads/writes(also called download/upload) algorithms. For the library librados method, we apply two different multi-threaded algorithms to optimize the files reads/writes. The results show that the performance of multi-threaded algorithms of downloading/uploading small files are improved by 7624%, 2827% respectively compared to the method of command line, the performance are improved by 58.95%, 170.36% compared to the method of library librados; the performance of one producer thread and one consumer thread model for downloading large files is improved by 22.42% compared to the method of downloading large files serially, but has no distinct effect on uploading large files. We analyze the reason and give suggestions for the future work.**

Keywords-Ceph, multi-threaded, producer/consumer, librados

I. Introduction

Improving the performance of the file systems is crucial to the overall performance of an tremendously broad class of applications. Traditional terms of settlement, represented by NFS[1], maintain the stateless server, a client transparently access files stored on a server. By using nonvolatile RAM, NFS accelerate writes[2]. Though NFS is widely used, the centralization architecture in the client/server framework has been proven an obvious hinder to the promotion of performance. Other storage solutions such as SAN(Storage-Area Networks)[3], NAS(Network-Attached Storage)[4], fail to find a balance between ease of use, performance and cost when the storage system scales up.

Recently, distributed file systems have resorted to object-based storage in which replace the hard disks with the intelligent Object Storage Devices(OSDs)[5] [6]. The OSDs promise significant improvements in scaling and administrative simplicity. An object storage device is the device that stores, retrieves and interprets those objects which encapsulate variable length user data and attributes of that data. The OSDs, hybrid between a disk and a file server, integrate a CPU, network interface and local storage services with fundamental magnetic disks or RAID[7].

Ceph, an open source, massively scalable, distributed file system that provides excellent performance, reliability, and scalability[8] [9]. The Ceph file system has three main modules: the client, each instance of which provides a POSIX-compliant distributed file system interface to a process or host computer[10]; a cluster of object storage devices(OSDs), which stores the entire data and metadata; a metadata server cluster(MDS), which supervises the file names and directories also termed namespace.

The methods that the users download files from Ceph and upload files to Ceph include command line and library librados. For the method of command line, it is difficult to improve the reads/writes performance given the specific Ceph storage cluster; For the method of library librados, files reads/writes will be accelerated with the help of the multi-threaded algorithms[11] [12]. Different multi-threaded algorithms are utilized for small files and large files. For small files, multiple threads read and write files simultaneously; For large files, one produer thread and one consumer thread model is designed to promote the reads/writes capability.

The producer/consumer model[13] [14] which is based on the Master/Slave pattern provide the capability to conduct multiple processes simultaneously while repeating at independent rates. The producer/consumer model is often employed when acquiring various sets of data to be handled in order. For example, compress/decompress the data blocks in the order they were read. Since compressing/decompressing one data block is much slower than the speed of reading/writing one data block, the producer/consumer model is very appropriate for this scenario[15]. We could also use this design pattern when optimize the speed of downloading/uploading files from/to Ceph.

The results show that the maximal speed of downloading/uploading small files concurrently reach to 85.74MB/S, 30.47MB/S respectively. Contrast to the serial speeds of 53.94MB/S, 11.27MB/S, the parallel algorithms improve the performance efficiently; For the large files, one producer thread and one consumer thread model improves the performance of downloading obviously, but parallel uploading and serial uploading have no obvious difference. We analyze the reason and give suggestions for the future work.

The paper is organized as follows: Section I introduces the distributed file system and multi-threaded algorithms; Section

II proposes the motivation of the work; Section III describes the detailed algorithms of uploading/downloading small/large files; Section IV provides the performance results and analysis; Conclusion are given in the section V.

II. Related Work And Motivation

Ceph is one of the most popular block storage or object storage backends for cloud platforms, Ceph will also shows the superior performance used for one file system. The reads/writes speed of Ceph is import in the above field. We investigate various methods of tuning performance of Ceph, the results show that no method optimizes the Ceph reads/writes taking the advantage of multi-threaded algorithms.

III. Algorithms and Implementations

A. Serial/Parallel Algorithms of Downloading/Uploading Small Files

Two methods of command line and library librados are utilized for the tasks of downloading/uploading small files. The details are described as follows.

1) Command Line: Client could access Ceph via the method of command line. Rados is a utility for interacting with a Ceph object storage cluster, part of the Ceph distributed storage system. Corresponding to downloading and uploading files, commands of 'rados get' and 'rados put' could fulfill the tasks respectively[16]. For example, 'rados get' reads the data from the cluster, the data saved in buffer are written to one local file; 'rados put' reads the data from local file, then the data are written to the Ceph storage cluster. We encapsulate the two commands in python scripts[17]. The algorithm 1 first connects to the Ceph storage cluster with the assistance of one configured cluster handle at line 1. Executes the command 'rados get' to read the data from Ceph storage cluster at line 8 or 'rados put' to write the data to the Ceph storage cluster at line 13. For downloading/uploading multiple files at a time of executing the program, the object names are stored in one local file in advance, then the next file will be downloaded or uploaded once the process of the front file is complete at line 5.

2) Librados: The rados.py modules is a Python wrapper for librados. Based on the RADOS[18], the librados API functions enables the clients to create their own interface to the Ceph storage system. The librados API functions provide the interface which contains two types of daemons in the Ceph storage cluster for the clients. The first is Ceph Monitor, which maintains a master copy of the cluster map; the other is Ceph OSD Daemon, which stores data as objects on a storage node. Based on the librados API, we implement serial/parallel algorithms to read or write data from/to the Ceph storage cluster.

The algorithm 2 shows downloading the small files from the Ceph storage cluster serially. Akin to the algorithm 1, connects to the Ceph storage cluster and opens the file which saves the objects' names representing the objects stored in the Ceph storage cluster. Reads all the data contained in one object at one time from Ceph to a buffer, then writes all the data stored

Algorithm 1 Serial Downloading/Uploading Small Files Based On Command Line

```
Input: file names list
Output: files downloaded/uploaded from/to Ceph
 1: cluster.open_ioctx              ▷ connect to the Ceph cluster
 2: open the file FILE_NAME  ▷ the file store the small file
    names which will be downloaded/uploaded from/to Ceph
 3: GOT            ▷ flag variable to determine the operation
 4: PUTTED
 5: while !EOF(FILE_NAME) do               ▷ loop till file end
 6:     line_of_file ← read(FILE_NAME)
 7:     if GOT == 1 then
 8:         rados get             ▷ download the file from Ceph
 9:         write the downloaded data to local file
10:     end if
11:     if PUTTED == 1  then
12:         read data from local file
13:         rados put                  ▷ upload the file to Ceph
14:     end if
15: end while
```

Algorithm 2 Serial Downloading Small Files

```
Input: file names list
Output: files downloaded from Ceph
 1: cluster.open_ioctx
 2: open the file FILE_NAME
 3: while !EOF(FILE_NAME) do
 4:     line_of_file ← read(FILE_NAME)
 5:     ioctx.read                ▷ download the file from Ceph
 6:     write the downloaded data to local file
 7: end while
```

in the buffer to one local file. Continues to read the next file until to the end of the file 'FILE_NAME'.

The algorihm 3 demonstrates downloading small files from the Ceph storage cluster concurrently. The main program assigns the tasks to every thread on average. The number of threads can be set to different value. Every thread orientates themselves to the different position of the file FILE_NAME based on the variant 'start_id' computed in advance, then downloads the files one by one. Multiple threads work independently, every thread downloads the files assigned to themselves at line 6 to 10.

The algorithm 4 uploads small files similarly to the operations of the algorithm 2, the difference between the two algorithms is that reading the local data to buffer, writing the data to the Ceph storage cluster with the help of cluster handle instead of reading from the Ceph storage cluster, writing to local disks.

The algorithm 5 uploads small files concurrently. The names of objects which will be uploaded are stored in the file 'FILE_NAME' at line 2. Calculates the quantity of files uploaded by every thread. The number assigned to the last thread is different from other threads if the variant 'THREAD_NUM' is not divisible by the amount of the files, the total amount

Algorithm 3 Parallel Downloading Small Files

```
Input: file names list
Output: all files downloaded from Ceph
 1: cluster.open_ioctx
 2: open the file FILE_NAME
 3: for i in range(THREAD_NUM) do
 4:     Download Threads Run
 5: end for
DownLoad Threads:
Input: FILE_NAME, start_id, numbers of files downloaded, ioctx
Output: partial files downloaded from Ceph
 6: for i range(numbers) do
 7:     line_of_file ← read(FILE_NAME) from start_id
 8:     ioctx.read
 9:     write the downloaded data to local file
10: end for
```

Algorithm 4 Serial Uploading Small Files

```
Input: file names list
Output: files uploaded to Ceph
 1: cluster.open_ioctx
 2: open the file FILE_NAME
 3: while !EOF(FILE_NAME) do
 4:     line_of_file ← read(FILE_NAME)
 5:     read the local file to buffer
 6:     ioctx.write                    ▷ upload the data to Ceph
 7: end while
```

Algorithm 5 Parallel Uploading Small Files

```
Input: file names list
Output: all files uploaded to Ceph
 1: cluster.open_ioctx
 2: open the file FILE_NAME
 3: for i in range(THREAD_NUM) do
 4:     Upload Threads Run
 5: end for
UpLoad Threads:
Input: FILE_NAME, start_id, numbers of files uploaded, ioctx
Output: partial files uploaded to Ceph
 6: for i range(numbers) do
 7:     line_of_file ← read(FILE_NAME) from start_id
 8:     read the local data to the buffer
 9:     ioctx.write
10: end for
```

of files subtract all the other numerical value assigned to the other threads, the residual value is assigned to the last thread. The previous details are not shown in the algorithm 5. The main program starts 'THREAD_NUM' threads, every thread uploads the files independently.

B. Serial/Parallel Algorithms of Downloading/Uploading Large Files

We implement the algorithms of downloading/uploading large files based on the method of library librados.

Algorithm 6 Serial Downloading Large Files

```
Input: file names list
Output: files downloaded from Ceph
 1: cluster.open_ioctx              ▷ connect to the Ceph cluster
 2: open the file FILE_NAME   ▷ the file store the large file
    names which will be downloaded from Ceph
 3: READ_SIZE                   ▷ define the size of block read
 4: while !EOF(FILE_NAME) do
 5:     line_of_file ← read(FILE_NAME)
 6:     Length ← ioctx.stat   ▷ compute the size of the large
    file uploaded
 7:     if ( Length % READ_SIZE == 0 ) then
 8:         end_block ← 0
 9:     else
10:         end_block ← 1
11:     end if
12:     BLOCKS_NUM = Length/READ_SIZE + end_block
13:     k ← 0
14:     while k < BLOCKS_NUM do
15:         k ← k+1
16:         if (k != BLOCKS_NUM) then
17:             read the data block from Ceph to the buffer ▷
    the size of the data block is READ_SIZE
18:             write the data block to the local file
19:         else
20:             read the last data block to the buffer
21:             write the data block to the local file
22:         end if
23:     end while
24: end while
```

The algorithm 6 shows that download large files serially. Different from small file, the large file is broken into several data blocks instead of processing the whole small file. The step of connecting to the Ceph cluster and opening the file which stores the names of files will be downloaded is the same as the above algorithm. Defines 'READ_SIZE' as the size of the data block at line 3 in the algorithm 6. The size of data block can be set to different numerical value. Different size of data block affect the download speed shown in the table II of section IV. Calculates the size of the file, the size is stored in the variant 'Length' at line 6. Determinates the number of data blocks at line 12. The number of files downloaded by the last thread may be different from the other thread, this circumstance is handled at line 16 to line 22. The algorithm invokes the librados API functions to download the data blocks one by one until the whole large file is downloaded. Repeat the above steps to download the next file.

Algorithm 7 Parallel Downloading Large Files

```
Input: file names list
Output: large files downloaded from Ceph
 1: cluster.open_ioctx              ▷ connect to the Ceph cluster
 2: while !EOF(FILE_NAME) do
 3:     Producer Thread Run
 4:     Consumer Thread Run
 5: end while
Producer Thread:
Input: ioctx, Large_File
Output: data blocks queue
 6: READ_SIZE                        ▷ define the size of block read
 7: Length ← ioctx.stat ▷ get the size of the object stored in
    Ceph
 8: if ( Length % READ_SIZE == 0 ) then
 9:     end_block ← 0
10: else
11:     end_block ← 1
12: end if
13: BLOCKS_NUM ← Length/READ_SIZE + end_block
14: while id < BLOCKS_NUM do
15:     id ← id+1
16:     if id != BLOCKS_NUM then
17:         data ← ioctx.read
18:         queue.put(data)
19:     else
20:         data ← ioctx.read           ▷ read the last data block
21:         queue.put(data)
22:     end if
23: end while
Consumer Thread:
Input: block_id
Output: large file downloaded from Ceph
24: open downloaded_file ▷ file handle, the file will store the
    downloaded data.
25: j ← 0
26: while j < BLOCKS_NUM do
27:     j ← j + 1
28:     data_get ← queue.get()
29:     downloaded_file.write(data_get)
30: end while
```

The algorithm 7 shows download large file based on one producer thread and one consumer thread model. As shown in the figure 1, we define two operations: `read` represents for reading one data block and pushing the data block into the buffering queue; `write` represents for poping out one data block from the buffering queue and writing the data block to local disks. The above explanation utilize the downloading procedure as the example to demonstrate the producer/consumer model. The uploading procedure based on the producer/consumer model is analogical to downloading. The details are omitted for briefness. In the algorithm 7, there are one producer thread for reads operation, one consumer thread for writes operation. The main program connects to the Ceph storage cluster, runs producer thread and consumer thread. In the producer thread, the data blocks are read and pushed into the queue in order; In the consumer thread, pops out one data block when the queue is not empty, writes the data block to local disks. The producer thread and the consumer thread orchestrate well with each other until all the data blocks downloaded, then the next file will be processed.

Algorithm 8 Serial Uploading Large Files

```
Input: file names list
Output: files uploaded to Ceph
 1: cluster.open_ioctx
 2: open the file FILE_NAME   ▷ the file store the large file
    names which will be uploaded to Ceph
 3: WRITE_SIZE       ▷ define the size of data block written
 4: while !EOF(FILE_NAME) do
 5:     line_of_file ← read(FILE_NAME)
 6:     Length ← os.path.getsize() ▷ compute the size of the
    large file uploaded
 7:     if ( length % WRITE_SIZE == 0 ) then
 8:         end_block ← 0
 9:     else
10:         end_block ← 1
11:     end if
12:     BLOCKS_NUM = Length/WRITE_SIZE + end_block
13:     k ← 0
14:     while k < BLOCKS_NUM do
15:         k ← k+1
16:         if (k != BLOCKS_NUM) then
17:             read the data block from local disk to the buffer
    ▷ the size of the data block is WRITE_SIZE
18:             ioctx.write
19:         else
20:             read the last data block to the buffer
21:             ioctx.write
22:         end if
23:     end while
24: end while
```

The algorithm 8 uploads large files to the Ceph storage cluster serially. The size of data block can be set to different value shown in the table IV, and the maximum size must be less than 95MB due to the limit of the Ceph storage cluster. The uploading procedure and downloading procedure are semblable between each other for the large files, the details of algorithm 8 are omitted.

The algorithm 9 deciphers uploading large files based on one producer thread and one consumer thread model as shown in the figure 1. The main program orchestrates the producer thread and the consumer thread with one buffering queue as an intermediate. The producer thread pushes the data blocks into the queue in sequence, the consumer thread pops the data blocks out and writes to the Ceph storage cluster by turn.

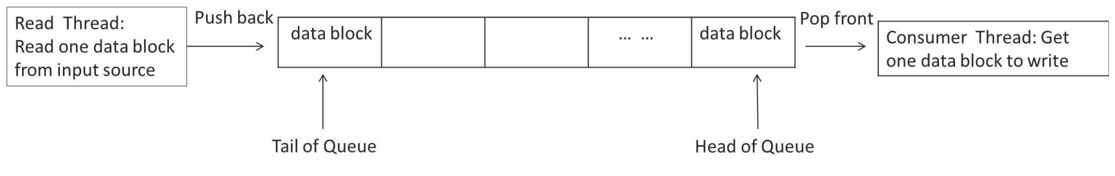
Read Thread: Read one data block from input source
Push back
data block
... ...
data block
Pop front
Consumer Thread: Get one data block to write
Tail of Queue
Head of Queue

Fig. 1. A demonstration of the producer/consumer model in downloading/uploading data blocks.

Algorithm 9 Parallel Uploading Large Files

```
Input: file names list
Output: large files uploaded to Ceph
 1: cluster.open_ioctx
 2: while !EOF(FILE_NAME) do
 3:     Producer Thread Run
 4:     Consumer Thread Run
 5: end while
Producer Thread:
Input: ioctx, Large_File
Output: data blocks queue
 6: READ_SIZE                        ▷ define the size of block read
 7: Length ← os.path.getsize()        ▷ get the size of the file
    uploaded to Ceph
 8: if ( Length % READ_SIZE == 0 ) then
 9:     end_block ← 0
10: else
11:     end_block ← 1
12: end if
13: BLOCKS_NUM ← Length/READ_SIZE + end_block
14: while id < BLOCKS_NUM do
15:     id ← id+1
16:     if id != BLOCKS_NUM then
17:         data ← read one data block from local file
18:         queue.put(data)
19:     else
20:         data ← read the last data block from local file
21:         queue.put(data)
22:     end if
23: end while
Consumer Thread:
Input: block_id
Output: large file uploaded to Ceph
24: j ← 0
25: while j < BLOCKS_NUM do
26:     j ← j + 1
27:     data_get ← queue.get()
28:     ioctx.write(data_get)
29: end while
```

IV. Performance Results and Analysis

A. Experiment environment

Four physical machines(4*Dell-R730xd) host the Ceph storage cluster in this paper. The detailed parameters of one machine described as follows: CPU: Intel(R) Xeon(R) E5-2630 v3, 2.40GHz; 16*3T disks; 128GB local memory. The Ceph storage system is deployed based on the above hardware. The software configuration: Ubuntu 14.04 operating system, the kernel version is 3.13.0; Ceph 0.94(3 monitors, 64 OSDs, 1 MDS). The Ceph was installed according to the Ceph documents[19]. Two sets of small files are used as test cases: the first set includes 130 files, the maximum size is 47MB, the minimum size is 24KB, the total size is 962MB ; the other set includes 3102 files, the maximum size is 146MB, the minimum size is 56KB, the total size is 24GB. For the large files, three files are used, the size are 614.03MB, 1363.18MB and 2726.36MB respectively as shown in the table II, III, IV, V.

B. Results and analysis

In the algorithm 1, we utilize the commands provided by the Ceph storage system to download/upload small files. Two sets of small files introduced in the section IV-A are used for measuring the performance of the algorithm 1. Average speed is calculated based on the two sets of files. The results show that the downloading speed is 1.11MB/S, the uploading speed is 1.02MB/S.

In the algorithm 2, we download small files serially from the Ceph cluster taking the advantage of the library librados. The results indicate that the speed is 53.94MB/S.

In the algorithm 4, serially upload small files to the Ceph cluster based on librados, the average speed is 11.27MB/S.

TABLE I
Parallel download/upload small files from/to Ceph(MB/S)

TNumber / SDOperations	2	4	8	10	16	32
SPD	64.82	74.06	**85.74**	77.89	75.87	75.63
SPU	17.38	22.35	25.69	**30.47**	27.79	30.31

[1] SDOperations: Speed of Different Operations.
[2] TNumber: the Number of Threads.
[3] SPD: Speed of Parallel Downloads, represents the speed of downloading small files from the Ceph cluster parallelly;
[4] SPU: Speed of Parallel Uploads, represents the speed of uploading small files to the Ceph cluster parallelly.

In the algorithm 3 and the algorithm 5, multi-threaded algorithms are used to improve the downloading/uploading performance. As shown in the table I, we utilize different numbers of threads(eg., 2, 4, 8, 10, 16, 32 respectively.) to measure the performance of downloading/uploading small files parallelly. The maximum speed for downloading small files is 85.74MB/S when the number of the threads is set to 8; the maximum uploading speed is 30.47MB/S when the number of the threads is set to 10. Contrast to the serial speed of downloading/uploading: 53.94MB/S, 11.27MB/S, the parallel algorithm improves the downloading/uploading(reads/writes) performance efficiently. Due to the low degree of parallelism(for example, the number of threads is 2), the speed is lower, though faster than the serial algorithm. When the number of threads get to the upper limit, the performance has no obvious improvement on account of the overload of threads maintenance(for example, the number is 32).

TABLE II
SERIAL DOWNLOAD LARGE FILES FROM CEPH(MB/S)

SBlock / SFiles	30MB	50MB	100MB	200MB	300MB
614.03MB	60.07	62.63	65.48	54.45	48.64
1363.18MB	61.70	65.09	64.83	55.11	49.52
2726.36MB	59.54	62.70	65.20	52.31	47.61

[1] SFiles: Size of Files, represents the file used for performance measurement.
[2] SBlock: Size of Block

TABLE III
PARALLEL DOWNLOAD LARGE FILES FROM CEPH(MB/S)

SBlock / SFiles	30MB	50MB	100MB	200MB	300MB
614.03MB	73.19	77.29	80.16	72.94	66.01
1363.18MB	76.80	80.08	79.99	71.80	72.98
2726.36MB	72.86	77.39	78.20	73.43	71.00

The algorithms 6 and 7 implement downloading large files serially, parallelly respectively. The performance results are shown in the table II and III. Three files of different size are used. For every file, different size of blocks(30MB, 50MB, 100MB, 200MB, 300MB) are set. Compare the table II to the table III, for every file, parallel algorithm improves the performance for every block size.

TABLE IV
SERIAL UPLOAD LARGE FILES TO CEPH(MB/S)

SBlock / SFiles	2MB	5MB	10MB	20MB	90MB
614.03MB	50.95	51.10	51.22	50.04	30.60
1363.18MB	54.41	54.65	54.34	53.61	29.58
2726.36MB	51.89	54.85	54.84	54.69	29.72

The table IV and V(corresponding to the algorithms 8 and 9) show the performance results of uploading large files serially and parallelly respectively. Different blocks size(2MB, 5MB, 10MB, 20MB, 90MB) are set. Limited by the Ceph storage cluster, the maximum block size is less than 95MB. Compare the table IV to V, for every file, the speed results have no obvious difference between serial and parallel algorithms for every block size. The reason is that the serial algorithm reads one data block, then writes the data block to Ceph, continues to write the next data block once the above data block has been processed; The producer thread in the parallel algorithm pushes the data block into the buffering queue in order, the consumer thread writes the data blocks to the Ceph storage cluster on by one. The performance enhancement are counteracted by the Ceph setting options. We intend to leverage multiple producer/consumer threads to improve the performance for the future work.

TABLE V
PARALLEL UPLOAD LARGE FILES TO CEPH(MB/S)

SBlock / SFiles	2MB	5MB	10MB	20MB	90MB
614.03MB	47.59	50.70	52.09	51.30	30.74
1363.18MB	50.23	53.63	53.45	53.72	30.51
2726.36MB	51.86	54.45	54.76	54.54	30.05

V. CONCLUSION

We implement the algorithms of Ceph downloading/uploading files via two methods: command line and library librados, and optimize the reads/writes speeds for the method of library librados by two multi-threaded algorithms.

We utilize large blocks of text to describe the algorithms. The reason is that if the data block is undersize, specially, one bit, the speed is too slow to fulfil some reads/writes tasks.

For the small files and large files, different multi-threaded algorithms are utilized. For the small files, the main program allocates almost the same amount of files to different threads, every thread processes different files, the results show that the performance of multi-threaded algorithms of downloading/uploading small files are improved by 7624%, 2827% respectively compared to the method of command line, the performance are improved by 58.95%, 170.36% compared to the serial method of library librados. The above performance results are calculated base on the results data 1.11MB/S, 1.02MB/S, 53.94MB/S, 11.27MB/S in the section IV-B and 85.74MB/S, 30.47MB/S in the table I. We intend to allocate the small files more evenly according to the size of files for the future work.

For the large files, we utilize one producer thread and one consumer thread model to optimize the reads/writes algorithms. The results show that the downloading performance is improved by 22.42%(Based on the results data: 65.48MB/S, 80.16MB/S as shown in the table II, III) compared to the method of downloading large files serially, but the parallel algorithm has no distinct effect on uploading large files. We intend to take the advantage of multiple producer/consumer threads model to optimize the files reads/writes algorithms for the future work.

ACKNOWLEDGMENT

The authors would like to thank the platform provided by Institute of Information Engineering, Chinese Academy of Sciences. The work was supported in part by Y5M0011A16.

References

[1] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, David Hitz. NFS Version 3:Design and Implementation. In Proceedings of the Summer 1994 USENIX Technical Conference, 137-151, 1994.

[2] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., Breaking Through the NFS Performance Barrier, Proceedings of the 1990 Spring European UNIX Users Group, Munich, Germany, Describes the application of nonvolatile RAM in solving the synchronous write bottleneck in NFS Version 2. 199-206, 1990.

[3] Jon Tate, Fabiano Lucchese, Richard Moore. Introduction to Storage Area Networks. IBM Redbooks. 2006.

[4] An Introduction to Network Attached Storage. HWM magazine, Jul 2003. ISSN 0219-5607. Published by SPH Magazines. 90-92.

[5] Brent Welch, Garth Gibson. Managing Scalability in Object Storage Systems for HPC Linux Clusters. In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies. 100-112, 2004.

[6] Schmuck, Frank, and Haskin, Roger. GPFS: A Shared-Disk File System for Large Computing Clusters. Proc First USENIX conf. on File and Storage Technologies (FAST02), Montery, CA Jan 2002.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 03), 2003.

[8] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. Proceeding OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation, 307-320, 2006.

[9] Sage A. Weil, CEPH: reliable, scalable, and high-performance distributed storage. A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in computer science. University of California Santa Cruz. 2007.

[10] POSIX Certification - Product details. get.posixcertified.ieee.org. Retrieved 2016-03-13.

[11] Borut Robic, Jurij Silc. A Survey of Processors with Explicit Multithreading. ACM Computing Surveys. 35(1):29-63, 2003.

[12] Lee, Edward A. The Problem with Threads. IEEE Computer. 39(5):33-42, May 2006.

[13] http://www.cs.cornell.edu/courses/cs3110/2010fa/lectures/lec18.html

[14] Daniel S, David L, Richard MY, Jeremy S, Christos K. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. International Conference on Parallel Architectures and Compilation Techniques, 20th. 2011.

[15] Ke Zhan, Chao Yang, Changyou Zhang, Jingjing Zheng, Ting W. PLDSRC: A Multi-threaded Compressor/Decompressor for Massive DNA Sequencing Data. 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science. 2014.

[16] http://docs.ceph.com/docs/hammer/rados/

[17] https://www.python.org/

[18] http://docs.ceph.com/docs/hammer/rados/api/python/

[19] http://docs.ceph.com/docs/master/start/