Journal of Information Processing Vol.25 438–447 (June 2017)

[DOI: 10.2197/ipsjjip.25.438]

Regular Paper

PPFS: A Scale-out Distributed File System for Post-petascale Systems

Fuyumasa Takatsu[1,†1,a)] Kohei Hiraga[1,b)] Osamu Tatebe[2,c)]

Received: October 24, 2016, Accepted: March 7, 2017

Abstract: The fusion of the research field of high-performance computing (HPC) with that of big data, which has become known as the field of extreme big data, is problematic in that file creation in storage systems such as distributed file systems is not optimized. That is, the large workload leads to simultaneous creations of many files by many processes when creating checkpoints. The need to improve the file creation processes prompted us to design a scale-out distributed file system for post-petascale systems named PPFS. PPFS consists of PPMDS, which is a scale-out distributed metadata server, and PPOSS, which is a scalable distributed storage server for flash storage. The high file creation performance of PPMDS was achieved by using a key-value store for metadata storage and non-blocking distributed transactions to update multiple entries simultaneously. PPOSS depends on PPOST, which is an object storage system that manages the underlying low-level storage, such as Fusion IO ioDrive, a flash device connected through PCI express supporting OpenNVM. The high file creation performance was attained by implementing the PPFS prototype using file creation optimization, termed bulk creation, to reduce the amount of communication between PPMDS and PPOSS. And, to enhance the I/O performance of PPOSS when the client process and PPOSS run on the same node, PPOSS accesses a local storage device directly. The prototype implementation of PPFS with a further file creation optimization called object prefetching achieves 138,000 Operations Per Second for file creation when using five metadata servers and 128 client processes, thereby exceeding the performance of IndexFS by 2.52 times. With local access optimization, PPOSS reached its limit at a block size of 16 KiB, which is an improvement of 1.5 times compared to before optimization. Furthermore, this evaluation indicates that PPFS has a good scalability on file creation and IO performance, that is required for post-petascale systems.

Keywords: distributed file system, metadata management, data management

1. Introduction

One of the main problems associated with extreme big data, which is a new area fusing high-performance computing (HPC) and big data, is that the creation of storage systems, such as distributed file systems, is not optimized. The area of extreme big data is noted for very large workloads that create many files by many processes at the same time for checkpointing purposes. On the other hand, the number of CPU cores employed in the HPC field has become increasingly large. In other words, an increasing number of processes create many files simultaneously. Therefore, it is important to improve the file creation performance. This trend is not expected to change in the future because the current trend in HPC for post-petascale systems is that computer nodes use accelerators such as GPU and MIC. Besides checkpointing, there are several create-intensive applications including gene sequencing, image processing, and phone and video logs, which require metadata operation performance such as file creations. Several studies have been devoted to overcoming this problem, e.g., PPMDS [1], GIGA+ [2], and IndexFS [3]. However, these authors attempted to address this problem by only improving the performance of the metadata server and they did not attempt to achieve a high IO performance, which is also important in the case of a distributed file system. Our aim is to realize a distributed file system with the ability to create 1 million files per second and access data at a rate of 100 TB/s. This requires us to re-design a scale-out distributed file system as a matter of critical importance.

The IO performance of a distributed file system depends on the local file system or local object storage that manages the underlying low-level storage. We proposed the design of object storage using OpenNVM [4] for HPC [5] (hereafter termed PPOST). This is used as the backend of a storage node that manages file data on a local storage device. It shows 740,000 Input/Output Operations Per Second (IOPS) for object creations that use 16 threads, which is 12 times faster than DirectFS [6], thereby reaching the performance limit of the device in terms of access performance. In this paper, we present our design of a scale-out distributed file system named PPFS, which uses the distributed metadata server PPMDS and a storage server using PPOST.

The contributions of this paper are as follows:

- We design a scale-out distributed file system for post-petascale systems
- We design two optimization techniques to improve the file creation performance

[1] Graduate school of System and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[2] Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[†1] Presently with Yahoo Japan Corporation
[a)] takatsu@hpcs.cs.tsukuba.ac.jp
[b)] hiraga@hpcs.cs.tsukuba.ac.jp
[c)] tatebe@cs.tsukuba.ac.jp

© 2017 Information Processing Society of Japan

- A prototype implementation achieves 138,000 ops/s for file creation when using five metadata servers and 128 client processes, and its performance exceeds that of IndexFS by 2.52 times
- The evaluation indicates that PPFS has a good scalability on file creation and IO performance that is required for post-petascale systems

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 introduces PPFS, a distributed file system for post-petascale systems, PPMDS, a distributed metadata server for post-petascale systems, and PPOSS, a scalable distributed storage server for post-petascale systems using OpenNVM. The prototype implementation of this system are described in Section 4 and are evaluated in Section 5. We conclude our work in Section 6.

2. Related Work

2.1 Distributed File System

This paper proposes a distributed file system for HPC named PPFS. A considerable amount of research about distributed file systems for HPC has been carried out.

The Panasas parallel file system [7] has a metadata management service named PanFS. This file system manages a file in multiple objects. When PanFS receives file creation requests, it not only presents object creation and write requests to the OSDs, but also maintains a local journal to record in-progress actions to recover from object creation failures and metadata server crashes. However, this design causes performance degradation. In our file system, a file is managed in an object. Moreover, PPFS only sends a request for object creation to a storage server. So, the current design of PPFS is inferior to PanFS in terms of fault tolerance in order to show higher performance.

Our distributed file system is designed to address file creation-intensive workloads. We show high file creation performance by using PPMDS, which optimizes the metadata service, including the namespace. However, many researchers have investigated the use of distributed metadata servers for HPC.

Lustre [8] supports the Lustre Distributed Namespace (DNE), which allows for the distribution of metadata on multiple metadata servers known as Luster Metadata Targets (MDT). Ceph [9], a distributed file system providing reliability and scalability, manages metadata on multiple metadata servers based on the server load. On the other hand, these metadata servers do not manage metadata on the key-value store. And, Lustre does not distribute the metadata in a single directory on multiple metadata servers. So, the metadata performance is limited to a single metadata performance. Moreover, Lustre and Ceph manage the metadata on remote storages. Storages and the metadata server do not always run on the same node. So, there is some overhead due to accessing remote storages via a network.

IndexFS [3], which is mentioned in Section 5, is a metadata server that manages metadata on a key-value store like PPMDS. IndexFS is middleware that adds support to existing distributed file systems such as PVFS [10], Lustre [8], and HDFS [11] to manage metadata and small files. It manages file and directory metadata of the same directory on multiple metadata servers by splitting large directories like GIGA+ [2]. This middleware is intended to improve the metadata performance of existing distributed file systems. So, the IO performance depends on these distributed file systems. Moreover, IndexFS dynamically splits a directory to several metadata servers when the number of files exceeds a threshold like GIGA+. When spliting the directory, the related metadata needs to be atomically transferred among metadata servers without any inconsistency. This atomic metadata transfer is left as an unsolved issue. PPMDS uses non-blocking transactions to resolve this issue. BatchFS [12] is an extension of IndexFS to reduce the RPC (Remote Procedure Call) overhead by using a relaxed consistency model. DeltaFS [13] is a serverless file system designed for exascale computing. However, it also uses other distributed file systems to store file data. ShardFS [14] is another distributed metadata service that shows high metadata performance. It replicates directory entries to all metadata servers, making it possible to obtain file metadata by using a single RPC. This design, on the other hand, requires a considerable amount of memory when billions of entries are created in post-petascale system. It even may cause memory shortage.

2.2 Storage System

We demonstrate high IO performance by achieving object storage using OpenNVM on PPOST. POSIX-compliant file systems, such as ext3 [15], ext4 [16], XFS [17], ZFS [18], and Btrfs [19], are often used as object storage for large-scale storage systems. For example, Ceph [9] used the OSD-based Btrfs, and Lustre [8] used the OSD-based ext4 or ZFS. SCMFS [20], NVMFS [21], F2FS [22], Yaffs [23], and JFFS2 [24] are designed for flash devices. However, these devices are not utilized for new flash primitives such as OpenNVM. Direct File System (DFS) [6] is a POSIX-compliant file system designed for flash devices supporting new flash primitives such as OpenNVM. However, the use of a POSIX-compliant file system as a local storage system makes it difficult to achieve the limit of the device. BlobSeer [25] and OBFS [26] are object storage systems. However, BlobSeer is a distributed data storage system and does not manage local storage devices directly. Instead, it uses local file systems to store data. OBFS is a local object storage system; however, it is not utilized for new flash primitives such as OpenNVM. Likewise, object-based SCM [27] is an object storage system supporting the OSD interface [28], but it is also not utilized for new flash primitives such as OpenNVM. Therefore, we decided to use PPOST as local storage backend.

3. PPFS: A Scale-out Distributed File System for Post-petascale Systems

We designed PPFS, —a scale-out distributed file system for post-petascale systems. The designed file system provides the following features:

(1) it shows high file creation performance by scaling the number of metadata servers
(2) it shows high file access performance by scaling the number of storage servers and accessing local object storage.

We realized the distributed file system named PPFS by using PPMDS, which is a scale-out distributed metadata server,

© 2017 Information Processing Society of Japan

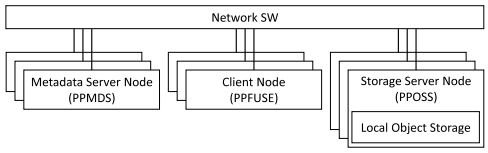


Fig. 1 Architecture of PPFS. This design is commonly used for distributed file systems such as Lustre.

and PPOSS, which is a scalable storage server. The architecture of PPFS is shown in **Fig. 1**. PPFS contains multiple metadata servers, storage servers, and clients on the same network. This design is commonly used for distributed file systems.

In this section, we describe the design of PPMDS, PPOSS, and PPFS in detail.

3.1 PPMDS: A Distributed Metadata Server for Post-petascale Systems

We use PPMDS [1], which is scale-out distributed metadata server, to show that a high file creation performance can be achieved. General file systems not only manage file data but also metadata such as namespaces, attributes, timestamps, the file size, and the inode number. The metadata management techniques of a traditional local file system are optimized for using local storage devices; thus, they are not suitable for using distributed file systems that manage metadata across multiple nodes. This is because the number of indirect references of inode block and data block increases when the file system grows bigger. In the case of post-petascale systems, when there are a large number of files in a specific directory, the directory entry becomes large that causes performance degradation. File metadata, such as inode, are managed easily because it involves one-to-one correspondence. However, a namespace, such as a file path, is difficult to manage because it has a tree structure. When file creation has been concentrated in a particular directory, parallelism is inhibited. Therefore, designing a metadata server for a distributed file system is highly challenging.

PPMDS [1] is a distributed metadata server for high-performance distributed file systems. Current parallel file systems are problematic in that the scalability is limited when file creation is concentrated in a particular directory. PPMDS is intended to solve this problem. PPMDS uses a key value store to manage file metadata and a list of metadata servers which manage the file in a specific directory. Every file has a parent directory. To manage file metadata on a key value store, PPMDS uses a pair of <inode number of the parent directory, filename> as a key of a file and stores its metadata as a value. With this technique, it becomes possible to look up the files in a specific directory using a range search based on its inode number. When using multiple metadata servers, the file metadata are distributed to multiple metadata servers. So, each metadata server also manages the list of metadata servers that manage the metadata entry of the directory. This list is managed for all metadata servers that manage the directory entry in these key value stores. When a client creates a file, the client obtains the inode number of the parent directory and sends a request to any metadata server. This server then starts the transaction of file creation and decides upon a metadata server to manage the metadata of the file, whereupon it requests the metadata server to create the file and commit the transaction. By using this technique, it is possible to create many files in a particular directory and to scale out the file creation performance.

3.2 PPOSS: A Scalable Distributed Storage Server for Post-petascale Systems

Next, we present the design of a storage server named PPOSS, which manages the file data. Our target system uses a flash storage device that supports OpenNVM flash primitives [4], including sparse address space and atomic batch write. OpenNVM version 0.7 enables us to use 144PB sparse address space regardless of the physical ioDrive capacity. The functionality of atomic batch write is to write or trim multiple blocks atomically.

3.2.1 PPOST—An Object Storage Using OpenNVM for High-performance Distributed File System

We previously proposed a local object storage that manages file data as an object on a local storage device using OpenNVM flash primitives for high-performance distributed file systems [5]. The sparse address space allows the design of an array of fixed-size regions that contain a single object, where the object can be addressed by a region number, i.e., object ID. Atomic batch write plays an important role in supporting the ACID properties in each write, and several optimizations. Each region manages object data. We proposed two object layouts, Version and Direct, in each region. Version Layout stores all the versions. The log to change data is appended with the commit block that has the version as the log-structured data format. Direct Layout stores only the latest version. Furthermore, we proposed two optimizations: one for updating the super region named Bulk Reservation, and another for initializing super blocks named Bulk Initialization. These optimizations achieved 740,000 IOPS using 16 threads, which is 12 times faster than DirectFS.

3.2.2 PPOSS

PPOSS is a distributed object storage server using PPOST as backend storage. PPOSS does not manage the metadata, such as the size, because these metadata are also managed in the metadata server. In addition, PPOSSs operate independently; in other words, there is no communication between different PPOSSs. Therefore, the IO performance of PPOSS is scalable.

PPOSS provides object creation, reading, writing, and deleting. Clients send requests for object creation to any PPOSS, and obtain a new object ID, including the server ID and object ID in the server. Clients use the object ID to access an object when reading or writing.

4. A Prototype Implementation

In this section, we provide a brief description of our implementation and optimization to achieve high IO performance.

4.1 PPFS

The prototype of PPFS is implemented by extending PPMDS to access PPOSS. PPMDS is composed of ppfuse, which is client software using FUSE [29], and ppmds, which is a metadata server. The combination of this software with PPOSS makes it possible to implement a distributed file system.

© 2017 Information Processing Society of Japan

PPMDS is written in C++, and it uses msgpack-rpc [30] for communication and it uses Kyoto Cabinet [31] as the key-value store for storing file metadata. Hence, PPOSS is also written in C++, and also uses msgpack-rpc for communication.

The method of file creation in PPFS is as follows:

(1) A client sends a file creation request to any metadata server.
(2) A metadata server that receives the request decides the inode number of the file.
(3) The metadata server which manages the metadata of the file decides the storage server ID by using this inode number as key by implementing Jump Consistent Hash [32].
(4) The metadata server requests an object creation to the storage server.
(5) The metadata server stores the metadata of the created file including the inode entry and the object ID.

4.1.1 Bulk Creation

In this prototype implementation, metadata servers, which request an object creation when requesting a file creation, are a client of storage servers. When these two servers are not in the same node, it is necessary to communicate over the network, which causes overhead. The backend local object storage of PPOSS supports the optimization of Bulk Initialization, which initializes the first blocks of N objects every N object creation requests. If the prototype implementation of the distributed file system follows the concept of these optimization techniques, the metadata server requests the storage server to create N objects for every N file creation requests. This optimization is called Bulk Creation. The file creation method with Bulk Creation is as follows:

(1) A metadata server prepares new object IDs by requesting the creation of N objects to all storage servers. These object IDs are stored in an associative array that uses the storage server ID as the key.
(2) A client requests a file creation to any metadata server.
(3) Metadata servers decide the inode number of the file.
(4) The metadata server which manages the metadata of the file decides the storage server ID by using this inode number as key by implementing Jump Consistent Hash [32].
(5) If the associative array does not have an object ID of the storage server, the metadata server requests the storage server to create N objects and adds it to the associative array.
(6) The metadata server obtains a new object ID from the associative array and deletes the object ID from the associative array.
(7) The metadata server stores the metadata of the created file including the inode entry and object ID.

Here, (1) is called during start-up time, and (2) to (7) are called during file creation. This procedure reduces the communication count between a metadata server and a storage server by a factor of N.

4.1.2 Object Prefetching

Bulk Creation reduces the communication count between the metadata server and the storage server to $1/N$. However, the file creation operation is blocked when creating objects. So, we also propose an optimization called Object Prefetching. This optimization uses a helper thread for object creation to hide the communication between the metadata server and the storage server. This obejct creation thread creates multiple objects when the number of object IDs kept by the metadata server are less than a threshold.

4.2 PPOSS

The implementation of PPOSS uses msgpack-rpc for communication. The data layout in a region is Direct Mode without Size to achieve a high access performance. Our object storage using OpenNVM for high-performance distributed file systems has two optimizations: one for updating the super region named Bulk Reservation, and another for initializing super blocks named Bulk Initialization. In this prototype implementation, we set parameters of Bulk Reservation and Bulk Initialization to 64. Thus, the super region, which manages the next region number, is updated after every 64 object creations and the super block, which is the first block of a region, is initialized every 64 times. Moreover, an object ID is an unsigned 64-bit integer: the first 32 bits are the storage server ID, and the last 32 bits are the object ID on the object storage server.

4.2.1 Direct Local Access

Some distributed file systems, such as Gfarm [33], Hadoop HDFS [11], and GFS [34], use the storage server as the compute node. This approach to writing data to the distributed file system entails data being written to the local storage server and enables a high IO bandwidth to be achieved. In addition, by cooperating with the job scheduling system, it is possible to achieve a high performance because the data movement on the network is reduced. When this characteristic is used on PPFS, the compute node also executes PPOSS when an object of a file is created on that PPOSS node. PPOSS uses msgpack-rpc for communication between the server and the client and this causes some memory copies. Thus, when the compute node also executes PPOSS, the client application writes directly to a storage device, thereby reducing the number of memory copies.

5. Evaluation

We evaluate the prototype implementation of the distributed file system designed in this paper. This section provides the methods used for these evaluations and the results that were obtained.

5.1 Evaluation Environment

The environment we evaluate our approach in is indicated in **Table 1**. We use from 1 to 5 nodes as metadata servers, from 1 to 8 nodes as clients, and from 1 to 10 nodes as storage servers. In addition, we use 128 client processes at most. Although these nodes are connected by InfiniBand, we use IPoIB (IP over InfiniBand).

5.2 File Creation Performance Evaluation

This evaluation measures the create performance of PPMDS by using mdtest HPC benchmark [35] version 1.9.3. The mdtest benchmark is an MPI-based metadata benchmark test program that performs create operations on directories and files in parallel. In this evaluation, we modified mdtest to access a different mount point in each process. At these mount points, the same file

© 2017 Information Processing Society of Japan

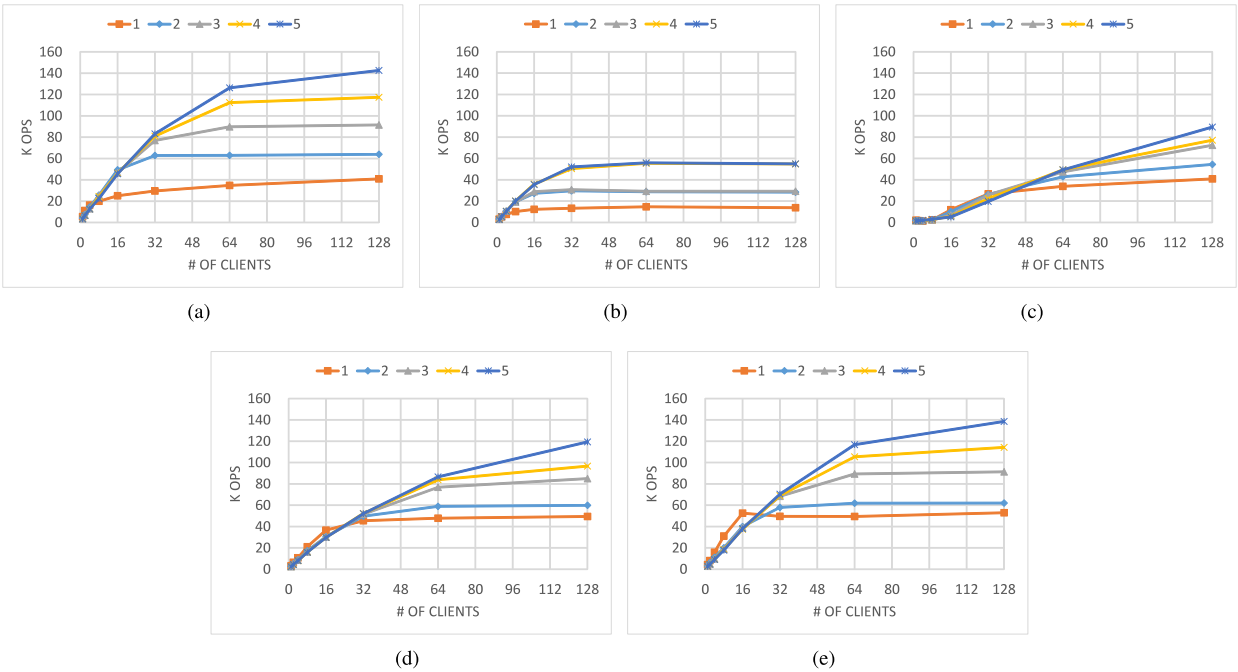


Fig. 2 File creation performance: (a) PPMDS (only metadata operations); (b) IndexFS; (c) PPFS; (d) PPFS with bulk creation ($N = 64$); (e) PPFS with object prefetching. The lines are the number of metadata servers.

Table 1 Node specification.

	Metadata Server	Client	Storage Server
CPU	Intel(R) Xeon(R) E5-2695 v2 2.40 GHz	Intel(R) Xeon(R) E5-2665 2.40 GHz	Intel(R) Xeon(R) E5620 2.40 GHz
# of cores	12	8	4
# of threads	12	8	8
# of sockets	2	2	2
RAM	64 GB		24 GB
OS	CentOS 6		
Network	Mellanox Technologies MT27500 4x FDR (56 Gbps)		Mellanox Technologies MT26428 x QDR (32 Gbps)
Storage Device	-	-	Fusion-io ioDrive 160 GB SLC
SDK	-	-	OpenNVM Version 0.7
# of nodes	1-5	1-8	1-10

system are mounted. This is because there is performance limitation of kernel for a single mount point. To avoid this limitation, we use multiple mount points.

This evaluation involves the creation of 5,000 files and 5,000 directories per mdtest process. The number of client processes we use ranges from 1 to 128; 640,000 files and 640,000 directories are created when using 128 client processes. We perform the evaluation five times and calculate the average of each number of client processes.

The results are shown in **Fig. 2**. In each graph in this figure, the horizontal and vertical axes show the number of processes and the number of operations per second.

Figure 2 (a), which shows the performance of the file creation procedure, shows that the performance is higher as the number of nodes increases. When five metadata servers and 128 client processes are used, the peak performance is 142,564.4 ops/s. The file creation performance exceeds the directory creation performance, because PPMDS does not create the list of file entries of the distributed server during file creation.

Next, for comparison purposes, we evaluate the performance of IndexFS [3], which also manages metadata on a key-value store such as PPMDS. Unlike IndexFS, many traditional distributed file systems focus on achieving a high data access performance rather than a high metadata performance. IndexFS is a middleware that adds support to existing distributed file systems such as PVFS [10], Lustre [8], and HDFS [11]. This key-value store manages metadata by using the pair consisting of the inode number and the hash value of the filename. According to a published report [3], IndexFS has been scaled to 128 metadata servers and shown an out-of-core metadata throughput that outperforms existing distributed file systems by 50% to two orders of magnitude. Therefore, we selected IndexFS with which to compare the performance of our PPMDS.

In Fig. 2 (b), which shows the performance of IndexFS, there is no difference between the performance achieved with two and three metadata servers or four and five metadata servers. This is because, depending on the number of files, IndexFS splits the directory to a power of two. When five metadata servers and 128 client processes are used, the performance of PPMDS is 2.60 times that of IndexFS. This is because, during file creation, PPMDS reads the distributed list of parent directories and writes the metadata to the metadata server, which manages the file entry. However, these read and write operations are not in conflict with each other, which means that there is no waiting time associated with either operation. PPMDS therefore shows a higher efficiency than IndexFS. But, IndexFS has a better scalability than PPMDS for a directory creation because all metadata servers of PPMDS store a list of nodes which manages files in its directory. Therefore, PPMDS doesn't have a good scalability for a direc-

© 2017 Information Processing Society of Japan

tory creation. However, we need a high file creation performance. Therefore, using PPMDS as a metadata server is better than using IndexFS.

Figure 2 (c) shows the performance of PPFS. It can be seen that five metadata servers achieve only 62.8% and 11.5% of the performance of PPMDS when using 128 and 16 client processes, respectively. This is because the response time increased due to the communication between MDS and OSS. Especially when using few clients, the effect is large.

Next, Fig. 2 (d) shows the performance of PPFS with bulk creation. In this evaluation, PPMDS communicates with PPOSS after the creation of every 64 files to create 64 objects. In this case, when using five metadata servers, PPFS achieves 119,373.9 ops/s. It is 83.7% and 65.2% of the performance of PPMDS for 128 and 16 client processes, respectively. Thus, the performance is improved up to 1.34 times compared with the performance of PPFS when using 128 clients, and 5.69 times when using 16 clients by the bulk creation optimization. This is because PPMDS communicates with PPOSS on every 64 file creation requests, which means that the number of communications is reduced. In addition, when using five metadata servers and 128 clients, the performance of PPFS exceeds that of IndexFS by 2.17 times. And also, this result indicates that PPFS has a very good scalability up to five nodes.

Figure 2 (e) shows the performance of PPFS with object prefetching. Object Prefetching is one of the optimizations for file creation that hides the communication between the metadata server and the storage server. When using five metadata servers, PPFS achieves 138,577.7 ops/s, and it is 97.2% of the performance of PPMDS. In addition, the performance of PPFS exceeds that of IndexFS by 2.52 times. Moreover, the metadata performance is improved linearly at the rate of 27.7 K OPS per server when increasing the number of metadata servers.

Our aim is to realize a distributed file system with the ability to create 1 million files per second and access data at a rate of 100 TB/s. PPFS has a good scalability and achieves 138,000 ops/s for file creation when using five metadata servers. PPFS metadata performance is improved almost linearly at the rate of 27.7 K OPS, which means 1 M OPS is expected to be achieved when using 37 metadata servers.

5.3 Remote File Access Performance Evaluation

This evaluation measures the write performance for each block size of our distributed file system.

We first evaluated the write performance of PPOSS as an initial evaluation, in which multiple client processes write to a single PPOSS simultaneously. Each client process wrote 128 MiB of data in each block size. When using 128 client processes, 16 GiB of data was written to PPOSS.

The result is shown in **Fig. 3**. The horizontal and vertical axes show the block size and the number of bytes written per second, respectively. The lines in Fig. 3 represent the performance with varying number of client processes. The performance is improved by increasing the number of client processes and reaches its limit at 32 processes. This is because of the performance limit of the storage device. We use the result obtained for 128 client processes as a baseline to evaluate the prototype implementation of the distributed file system based on the combination of PPMDS and PPOSS.

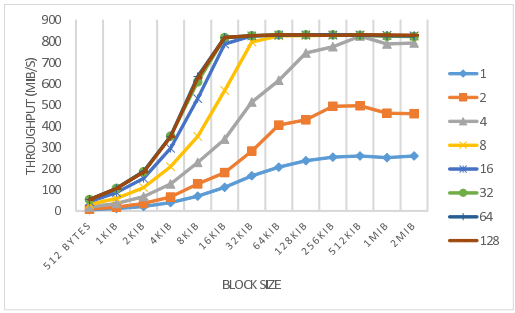


Fig. 3 Write performance of PPOSS. The lines represent the performance with varying number of client processes.

Next, we evaluated the write performance of the prototype implementation of the distributed file system by combining PPMDS and PPOSS for each block size. We use IOR HPC Benchmark [36] version 2.10.3, an MPI program, which writes data of a specific size to a file in parallel. In this evaluation, each process wrote 128 MiB of data with varying block sizes and numbers of metadata servers. The varying number of IOR processes ranges from 1 to 128. We used ppfuse, which is the client application of our distributed file system, to mount 128 mount points spread across eight nodes. Each of the IOR processes uses a different mount point.

The result is shown in **Fig. 4** (a) and Fig. 4 (b), which differ in terms of the number of metadata servers. The horizontal and vertical axes show the block size and the number of write bytes per second, respectively. The lines in Fig. 4 show the performance with varying numbers of IOR processes.

This evaluation did not include IndexFS, because IndexFS is a middleware that is executed on other distributed file systems, such as Lustre, PVFS, HDFS. Thus, the I/O performance of IndexFS depends on these distributed file systems.

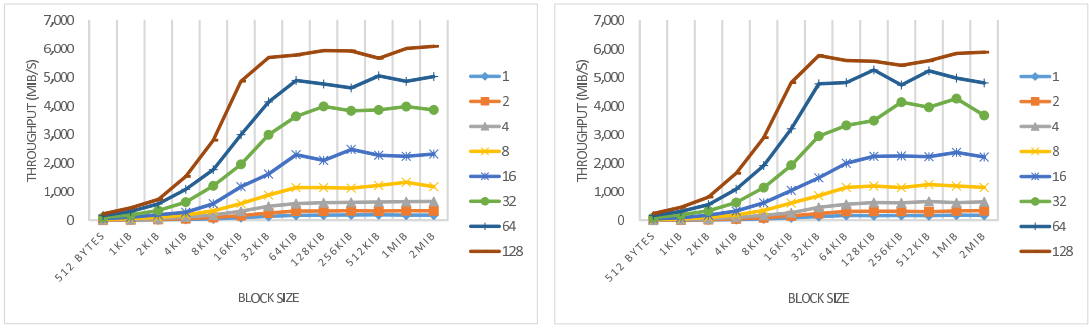
A comparison of the graphs in Fig. 4 indicates that the effect of the number of metadata servers is not large. This is because the metadata server is accessed only when calling open() and close() when writing to a file to get the object ID and update the file size. Thus, the metadata server is not accessed during each write() operation.

Peak performance is achieved at 6,085.75 MiB/s when using one metadata server, 128 client processes, and a block size of 2 MiB. Under these conditions, each storage server is accessed at 608.6 MiB/s on average. However, the initial performance evaluation shown in Fig. 3 indicates that the performance is 828.2 MiB/s under the same conditions. Thus, the PPOSS access performance used in PPFS undergoes 26.5% degradation compared to the single node performance of PPOSS.

We investigated the cause by evaluating the performance using one PPMDS node and 128 client processes by changing the number of PPOSS nodes. In this evaluation, we also used IOR HPC Benchmark [36] version 2.10.3, with each process writing a 128 MiB file in each block size.

The result is shown in **Fig. 5**, in which the horizontal and vertical axes show the block size and the number of bytes written per

© 2017 Information Processing Society of Japan



(a) (b)

Fig. 4 Write performance for varying numbers of metadata servers: (a) one and (b) five metadata servers. The lines show performance with varying numbers of IOR processes.

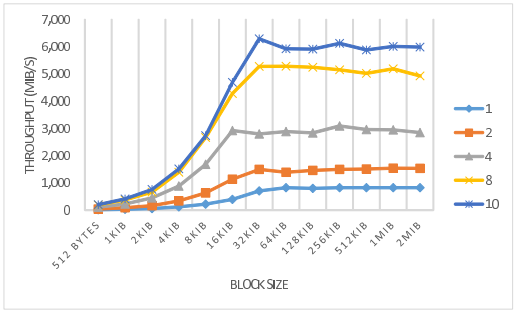


Fig. 5 Write performance by changing the number of PPOSS nodes. The lines show performance with varying numbers of PPOSS nodes.

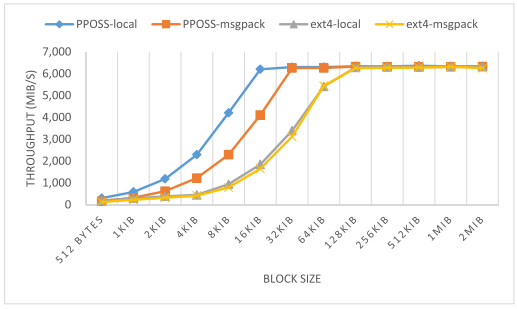


Fig. 6 Local access performance by changing the block size.

second, respectively. The lines in Fig. 5 show the performance with varying numbers of IOR processes.

Figure 5 shows that the overall performance of our distributed file system increases as a function of the number of PPOSS nodes. Compared to the ideal performance given the performance of a single PPOSS (829 MiB/s), the actual result is about 24% worse (e.g., 629 GiB/s instead of 6,285 GiB/s with 10 nodes). This is because the files are not evenly distributed throughout the PPOSS nodes and the total number of client processes is few. So, some PPOSSs are accessed by few clients. When using 10 PPOSS nodes and 128 clients, it is 13 clients at most for each node even if it is cleanly distributed. To show a high performance, each node needs more clients. Moreover, the performance is degraded when a small block size is used. This is because the ppfuse uses FUSE and msgpack-rpc. FUSE and msgpack-rpc need memory copies. When a small block size is used, the effect of overhead due to memory copy increases.

5.4 Local File Access Performance Evaluation

We evaluated the file access performance when the file data is managed on the node the client process is running on. For this evaluation we also used IOR HPC Benchmark [36] version 2.10.3. The parameters are the same as in Section 5.3. We used one PPMDS node, and eight PPOSS nodes, which also execute client processes.

Figure 6 shows the result of the local access performance evaluation involving eight PPOSS nodes and 128 client processes. In this case, each node executes 16 client processes. The blue line (PPOSS-local) shows the performance of PPOSS with Direct Local Access optimization enabled, and the orange line (PPOSS-msgpack) shows the performance of PPOSS with Direct Local Access optimization disabled. Furthermore, to evaluate the PPOST performance, we implemented PPOSS using ext4 as backend storage systems. The gray line (ext4-local) shows the performance of PPOSS using ext4 with Direct Local Access optimization enabled, and the yellow line (ext4-msgpack) shows the performance of PPOSS using ext4 with Direct Local Access optimization disabled.

In Fig. 6, all the methods reached their limits at 6,200 MiB/s, which is close to the performance limit of the device. However, PPOSS-local reached the limit using a 16-KiB block, at which point it was 1.5 times faster than PPOSS without the optimization. Thus, the optimization is effective in terms of achieving a high write performance. In addition, the performance of PPOSS-local is 3.4 times more efficient than ext4-local. This confirms that the use of PPOST as a backend storage system is effective.

Next, we evaluated the scalability of PPOSS by using IOR. The parameters of IOR are the same as in the previous evaluation. **Figure 7** shows the result of the evaluation for different block sizes. In Figs. 7 (a)–(d), the horizontal axis of each plot shows the number of PPOSS nodes and client processes. As each PPOSS node executes 16 client processes, the total number of client processes are 128 when eight PPOSS nodes are used.

Figure 7 (a) shows the result when a small block size of 512 bytes is used. This result indicates that PPOSS has a good scalability. When eight PPOSS nodes are used, the performance of PPOSS-local exceeds that of PPOSS-msgpack by 1.9 times. This is because the optimization reduces the number of memory copy operations. Further, ext4-local is 1.2 times more efficient than PPOSS-msgpack. This is because the overhead due to memory copy is larger than the overhead caused by using ext4 as backend

© 2017 Information Processing Society of Japan

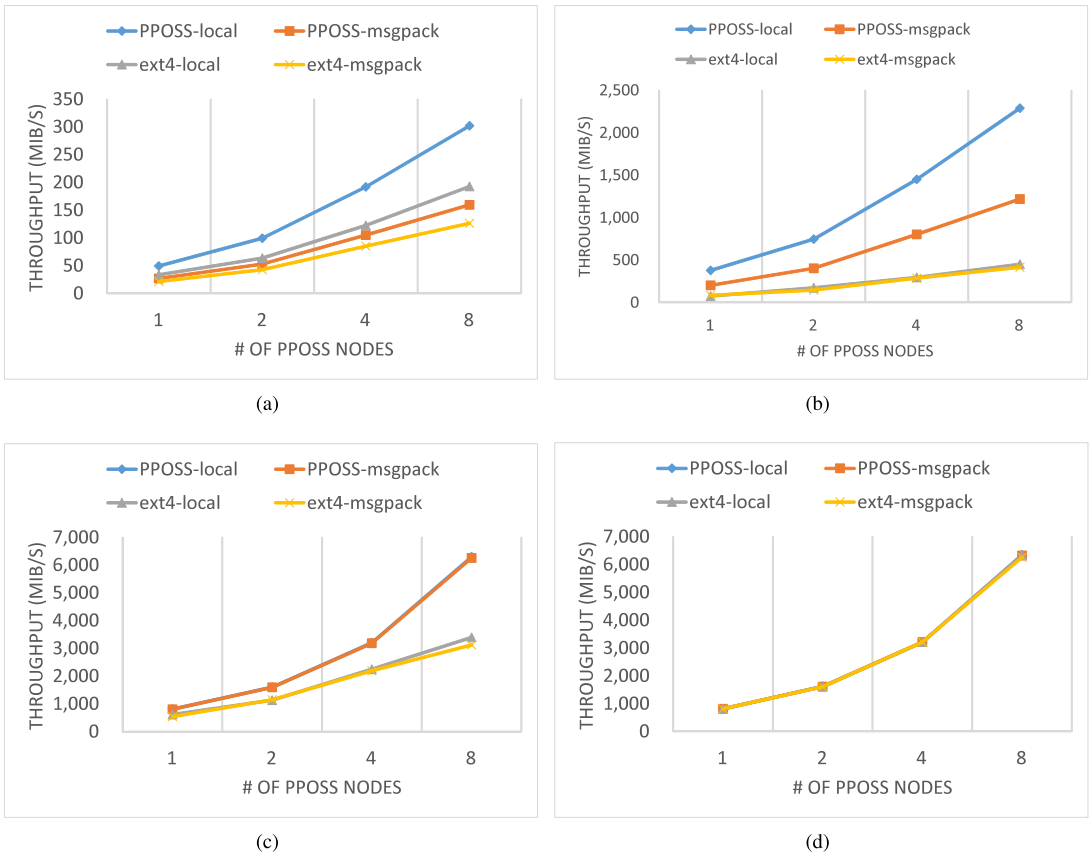


Fig. 7 Local access performance for a different number of PPOSS nodes: (a) small block size of 512 B; (b) medium block size of 4 KiB; (c) large block size of 32 KiB; (d) very large block size of 2 MiB.

when using a small block size. Figure 7 (b) shows the result for a medium block size of 4 KiB, also indicating that PPOSS has a good scalability. When eight PPOSS nodes are used, PPOSS-local shows an improvement of 1.9 times compared to PPOSS-msgpack and ext4-local is 1.1 times faster than ext4-msgpack. The increase in performance of PPOSS using ext4 as backend is smaller than when using PPOST. This is because, when using ext4, the overhead is larger than the overhead caused by a memory copy. Figure 7 (c) shows the result that was obtained for a large block size of 32 KiB. This result also shows that PPOSS has a good scalability. The performance of PPOSS-local when eight PPOSS nodes are used is almost the same as for PPOSS-msgpack. This is because, by using a large block size, the time taken for memory copying is sufficiently smaller than the time it takes to write data, so the performance is almost the same. The performance of PPOSS-local exceeds that of ext4-local by 1.9 times. This means that the use of PPOST as a backend storage system is effective. Figure 7 (d) shows the result for a very large block size of 2 MiB. This result also confirms the scalability of PPOSS. In this case, the use of eight PPOSS nodes almost produces the same result for all local access modes. This is because the use of a very large block size together with ext4 reduces the overhead caused by a memory copy. Moreover, the IO performance is improved linearly at the rate of 775 MiB/s per server when increasing the number of storage servers.

Our aim is to realize a distributed file system with the ability to create 1 million files per second and access data at a rate of 100 TB/s. PPFS IO performance is improved almost linearly at the rate of 775 MiB/s, which means 100 TB/s is expected to be achieved when using 166,000 storage servers.

6. Conclusion

This paper presents the design of PPFS—a scale-out distributed file system for post-petascale systems. PPFS consists of PPMDS, which is a scale-out distributed metadata server, and PPOSS, which is a scalable distributed storage server for flash storage. High file creation performance of PPMDS was achieved by using a key-value store for metadata storage and non-blocking distributed transactions to update multiple entries simultaneously. PPOSS depends on PPOST, which is an object storage system that employs OpenNVM. The use of OpenNVM markedly improves the object-creation and the access performance of PPOST. A high file creation performance was attained by implementing the PPFS prototype using file creation optimization, termed bulk creation, to reduce the amount of communication between PPMDS and PPOSS. In addition, the I/O performance of PPOSS is enhanced when the client process and PPOSS run on the same node, because this avoids unnecessary memory copy.

In the file creation evaluation, PPMDS outperformed IndexFS by 2.6 times when using five metadata servers, eight client nodes, and 128 client processes. Further, when five metadata servers were used, PPFS with object prefetching achieves 138,000 ops/s. Moreover, the performance of PPFS exceeds that of IndexFS by 2.52 times.

In terms of remote access evaluation, we showed that the access performance does not depend on the number of metadata

© 2017 Information Processing Society of Japan

servers. With local access optimization, PPOSS reached its limit at a block size of 16 KiB, which is an improvement of 1.5 times compared to before optimization and it is enhanced 3.4 times compared to PPOSS using ext4 as backend instead of PPOST. Thus, the results indicate that the optimization using PPOST as a backend storage system is effective.

Furthermore, these evaluations indicate that PPFS has a good scalability on the file creation and the IO performance, that is required for post-petascale systems.

Future work includes the simulation on a larger number of clients and designing fault tolerance systems.

Acknowledgments This work is supported by JST CREST, "System Software for Post Petascale Data Intensive Science", "EBD: Extreme Big Data—Convergence of Big Data and HPC for Yottabyte Processing" and "Statistical Computational Cosmology with Big Astronomical Imaging Data".

References

[1] Hiraga, K. and Tatebe, O.: Design and implemantation of distirbuted metadata server using Non-blocking transaction for distributed file system, *IPSJ SIG Technical Reports of High Performance Computing (HPC)*, Vol.2012, No.28, pp.1–9 (2012) (in Japanese).
[2] Patil, S. and Gibson, G.A.: Scale and Concurrency of GIGA+: File System Directories with Millions of Files., *Proc. 9th USENIX Conference on File and Storage Technologies*, Vol.11, pp.177–190 (2011).
[3] Ren, K., Zheng, Q., Patil, S. and Gibson, G.: IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.237–248 (2014).
[4] Fusion-io: NVM Primitives Library, available from ⟨http://opennvm.github.io/nvm-primitives-documents/⟩ (2014).
[5] Takatsu, F., Hiraga, K. and Tatebe, O.: Design of Object Storage Using OpenNVM for High-performance Distributed File System, *Journal of Information Processing*, Vol.24, No.5 (2016).
[6] Josephson, W.K., Bongo, L.A., Li, K. and Flynn, D.: DFS: A File System for Virtualized Flash Storage, *ACM Trans. Storage*, Vol.6, No.3, pp.14:1–14:25 (2010).
[7] Welch, B., Unangst, M., Abbasi, Z., Gibson, G.A., Mueller, B., Small, J., Zelenka, J. and Zhou, B.: Scalable Performance of the Panasas Parallel File System, *Proc. 6th USENIX Conference on File and Storage Technologies*, Vol.8, pp.1–17 (2008).
[8] Koutoupis, P.: The Lustre Distributed Filesystem, *Linux Journal*, Vol.2011, No.210 (2011).
[9] Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E. and Maltzahn, C.: Ceph: A scalable, high-performance distributed file system, *Proc. 7th Symposium on Operating Systems Design and Implementation*, pp.307–320 (2006).
[10] Ross, R.B., Thakur, R. et al.: PVFS: A parallel file system for Linux clusters, *Proc. 4th Annual Linux Showcase and Conference*, pp.391–430 (2000).
[11] Shvachko, K., Kuang, H., Radia, S. and Chansler, R.: The hadoop distributed file system, *Proc. 2010 IEEE 26th Symposium on mass Storage Systems and Technologies*, pp.1–10 (2010).
[12] Zheng, Q., Ren, K. and Gibson, G.: BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers, *Proc. 9th Parallel Data Storage Workshop*, pp.1–6 (2014).
[13] Zheng, Q., Ren, K., Gibson, G., Settlemyer, B.W. and Grider, G.: DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers, *Proc. 10th Parallel Data Storage Workshop*, pp.1–6 (2015).
[14] Xiao, L., Ren, K., Zheng, Q. and Gibson, G.A.: ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems, *Proc. 6th ACM Symposium on Cloud Computing*, pp.236–249 (2015).
[15] Ts'o, T.Y. and Tweedie, S.: Planned Extensions to the Linux Ext2/Ext3 Filesystem, *Proc. FREENIX Track: 2002 USENIX Annual Technical Conference*, pp.235–243 (2002).
[16] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A. and Vivier, L.: The New ext4 Filesystem: Current Status and Future Plans, *Proc. 2007 Linux Symposium*, pp.21–33 (2007).
[17] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M. and Peck, G.: Scalability in the XFS File System, *Proc. USENIX 1996 Annual Technical Conference*, pp.1–14 (1996).
[18] Bonwick, J. and Moore, B.: ZFS: The last word in file systems (2007).
[19] Rodeh, O., Bacik, J. and Mason, C.: BTRFS: The Linux B-tree filesystem, *ACM Trans. Storage*, Vol.9, No.3, pp.9:1–9:32 (2013).
[20] Wu, X. and Reddy, A.L.N.: SCMFS: A File System for Storage Class Memory, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.39:1–39:11 (2011).
[21] Qiu, S. and Reddy, A.: NVMFS: A hybrid file system for improving random write in nand-flash SSD, *Proc. IEEE 29th Symposium on Mass Storage Systems and Technologies*, pp.1–5 (2013).
[22] Lee, C., Sim, D., Hwang, J. and Cho, S.: F2FS: A New File System for Flash Storage, *Proc. 13th USENIX Conference on File and Storage Technologies*, pp.273–286 (2015).
[23] Aleph One: YAFFS: Yet Another Flash File System, available from ⟨http://www.yaffs.net/⟩ (2002).
[24] Woodhouse, D.: JFFS: The journalling flash file system, *Ottawa Linux Symposium* (2001).
[25] Nicolae, B., Antoniu, G., Bougé, L., Moise, D. and Carpen-Amarie, A.: BlobSeer: Next Generation Data Management for Large Scale Infrastructures, *Journal of Parallel and Distributed Computing*, Vol.71, No.2, pp.168–184 (2011).
[26] Wang, F., Brandt, S.A., Miller, E.L. and Long, D.D.E.: OBFS: A File System for Object-based Storage Devices, *Proc. 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp.283–300 (2004).
[27] Kang, Y., Yang, J. and Miller, E.L.: Object-based SCM: An Efficient Interface for Storage Class Memories, *Proc. 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, pp.1–12 (2011).
[28] Weber, R.: SCSI Object-Based Storage Device Commands (2004).
[29] FUSE: Filesystem in Userspace, available from ⟨http://fuse.sourceforge.net/⟩.
[30] msgpack-rpc, available from ⟨https://github.com/msgpack/msgpack-rpc⟩.
[31] FAL Labs: Kyoto Cabinet: A straightforward implementation of DBM, available from ⟨http://fallabs.com/kyotocabinet/⟩.
[32] Lamping, J. and Veach, E.: A fast, minimal memory, consistent hash algorithm, *arXiv preprint arXiv:1406.2294* (2014).
[33] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol.28, No.3, pp.257–275 (2010).
[34] Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google File System, *Proc. 19th ACM Symposium on Operating Systems Principles*, pp.29–43 (2003).
[35] mdtest HPC Benchmark, available from ⟨http://mdtest.sourceforge.net/⟩.
[36] IOR HPC Benchmark, available from ⟨https://sourceforge.net/projects/ior-sio/⟩.

Fuyumasa Takatsu received a Ph.D. in Engineering from University of Tsukuba in 2017. His research interest is distributed file system.

Kohei Hiraga is a Ph.D. candidate at University of Tsukuba. He received his M.E. from University of Tsukuba in 2011. Main research interests are grid computing and distributed file system.

© 2017 Information Processing Society of Japan

Osamu Tatebe received a Ph.D. in computer science from the University of Tokyo in 1997. He worked at Electrotechnical Laboratory (ETL), and National Institute of Advanced Industrial Science and Technology (AIST) until 2006. He is now a professor in Department of Computer Science at University of Tsukuba. He has been a co-chair of Grid File System WG of Open Grid Forum since 2004. His research area is high-performance computing, data-intensive computing, parallel and distributed system software. He is a member of ACM and Japan Society for Industrial and Applied Mathematics (JSIAM).

© 2017 Information Processing Society of Japan