These methods are for physical computers and assume that manages utilization (free or used) of blocks in an HDD. In addition, the other is that each zone in an HDD has different sequential I/O performance. The proposed methods control the location to store intermediate data by modifying block bitmap of filesystem, which manages utilization (free or used) of blocks in an HDD. In addition, we propose striping layout for applying these methods for virtualized environment using image files. We then present performance evaluation of the proposed method and demonstrate that our methods improve the Hadoop application performance.

key words: Hadoop, big data, HDD, filesystem

1. Introduction

Hadoop by ASF (Apache Software Foundation) is a popular open-source MapReduce implementation [1] for processing large-scale data, which is called big data. MapReduce jobs are composed of Map phase and Reduce phase. A Map task performs filtering and sorting of a chunk of data, and creates intermediate data. Reduce tasks summarize all output data of Map tasks and create the final results. In the cases of jobs, wherein huge scale of output files of all relevant Map tasks are transmitted into Reduce tasks, such as TeraSort, the Reduce tasks are the bottleneck tasks and are I/O bounded for processing many large output files. In most cases, including TeraSort, the intermediate data, which include the output files of the Map tasks, are large and accessed sequentially. For improving the performance of these jobs, it is important to increase the sequential access performance. In this paper, we propose methods for improving the performance of Reduce tasks of such jobs by considering the following two things. One is that these files are accessed sequentially on an HDD, and the other is that each zone in an HDD has different sequential I/O performance. The proposed methods control the location to store intermediate data by modifying block bitmap of filesystem, which manages utilization (free or used) of blocks in an HDD. In addition, we propose striping layout for applying these methods for virtualized environment using image files. We then present performance evaluation of the proposed method and demonstrate that our methods improve the Hadoop application performance.

2. MapReduce

MapReduce is a programming model proposed by Dean et al. [5], and its process is composed of a Map phase, which processes a key-value pair to generate a set of intermediate key-value pairs, and a Reduce phase, which merges all intermediate values associated with the same intermediate keys. The overview of MapReduce data processing is illustrated in Fig. 1.

In the Map phase, JobTracker splits input data in the HDFS into chunks and allocates the chunks to Map tasks. The Map tasks are allocated to TaskTrackers. A TaskTracker creates key-values pairs from allocated chunks. It then applies the Map function to each key-value pair and creates output key-value pairs. The output key-value pairs are stored in intermediate files.

In the Reduce phase, JobTracker shuffles intermediate data based on keys and allocates these to Reduce tasks. The Reduce tasks are allocated to the TaskTracker, and it performs the Reduce function on the given intermediate data. Then, the final results are obtained.

In the Reduce phase, the intermediate data from all Map tasks are transferred to the nodes that are allocated Reduce tasks, and are processed in the nodes. In many cases, the disk I/Os in Reduce phase are bottlenecks because this
phase causes huge amount of I/Os, such as read and write processes for transmitting and receiving intermediate data, read processes for input data of Reduce tasks, and write processes for output data of Reduce tasks. In addition, these data are managed with large files whose sizes are 64MB to 256MB on HDFS and are accessed sequentially on the disk. Therefore, we assume that improving sequential I/O performance is important for increasing the Hadoop performance.

3. Analysis of Behavior

3.1 Experimental Setup

We executed TeraSort and analyzed the system behavior for investigating its bottleneck process in a physical environment by measuring CPU and I/O utilization using the vmstat and iostat commands. The experimental environment is illustrated in Fig. 2. There is one master node and four slave nodes. The sizes of the input data are 16 GB and 32 GB with single-Reduce-tasked and multiple-Reduce-tasked jobs, respectively. The specifications of the master node and the slave nodes are shown in Table 1 and Table 2, respectively. Hadoop version is 2.0.0-cdh4.2.1. HDFS block size is 64 MB. All the Hadoop data, including intermediate data, are stored in a 500 GB HDD with Ext3 filesystem format. The specification of the HDD for Hadoop data is described in Table 3.

3.2 Bottleneck

In this subsection, we explore the bottleneck issue of the jobs. The transition of CPU utilization and I/O wait ratio of the Reduce node of a single-Reduce-tasked job are depicted in Fig. 3. I/O utilization of the node is shown in Fig. 4. Those of a non-Reduce node are shown in Fig. 5 and Fig. 6. The horizontal axis in every graph represents time from the beginning of the execution. The vertical axes show ratio of CPU usage, CPU I/O wait, and I/O usage. In this experiment, the last Map task finished at 506 sec. Thus,
only the Reduce tasks were running after the time. Figure 3 and Fig. 4 show that the I/O utilization of the Reduce node was kept at 100% for almost the entire time. This implies that disk I/O in the Reduce node is the bottleneck of the job. This is mainly because the single Reduce node receives many intermediate data from all the Map tasks and writes/reads these to/from the HDD.

Figure 5 and Fig. 6 imply non-Reduce nodes are not the bottleneck of the job. After finishing Map tasks, their loads were always low. Even before finishing, their loads were not high. That is, I/O utilization is not high at most times and CPU utilization is less than I/O utilization in Fig. 4. In the non-Reduce node, disk accesses are conducted mainly for reading input data. Comparing Fig. 4 and Fig. 6 indicates that the disk I/O load in non-Reduce node is relatively low.

From the figures and above observation, we can see that the single Reduce task in the Reduce node accounts for a large part in the entire running time and the Reduce task is in the I/O bound state.

Figure 7 and Fig. 8 show CPU and I/O utilization of a node in the multi-Reduce-tasked job. The number of Reduce task is four, then each slave node executes a Reduce task. In this experiment, the last Map task finished at 2,060 sec. Thus, only the Reduce tasks were running after the time. These figures show that the I/O utilization was almost always kept at 100% for processing Reduce tasks. This implies that disk I/O in the Reduce phase is the bottleneck of the job. This is mainly because each node receives many intermediate data from all the Map tasks and writes/reads these to/from the HDD. Similar to the single-Reduce-tasked job, Reduce phase is I/O bounded.

3.3 I/O Size

In this subsection, we investigate sizes of the I/O requests in the Reduce node. We modified the implementation of the SCSI subsystem in the Linux kernel in order to record I/O requests issued for the HDD. The modified kernel stores the size, an accessed address, and time of every I/O request. Because the monitoring is performed in the SCSI layer, which is lower than the page cache in block layer, the monitored requests are actually issued ones.
Figure 9 and Fig. 10 depict the relation between sizes of issued requests and their frequencies. “Merged I/O size” in the figure is obtained by merging temporally and spatially continuous I/O requests. For example, three temporally continuous requests for address 128 KB with size 16 KB and one for address 144 KB with size 16 KB, and one for address 160 KB with size 16 KB are consolidated into a request with size 48 KB. Most operating systems, including Linux, divide a large I/O request from an application into small I/O requests, and send these to a storage device one by one. Focusing on HDD performance, merged I/O size is important because this size determines HDD behavior, i.e., sequential access or random access.

The figures indicate that many large I/O requests are issued to the HDD in the Reduce phase. This implies that the HDD is working highly sequentially. As a result, we can conclude that improving sequential access performance is desired for decreasing time to complete the job.

### 3.4 Performance Improvement

Some studies demonstrated that discussions on disk drive characteristics could improve I/O performance [6]–[8]. Higher sequential access performance can be obtained in outer zones of the HDD [6] and utilizing these outer zones is important for improving sequential access performance. In addition, allocating related data on disk trace boundaries enables avoiding most rotational latency and track crossing overheads [7] and allowing the spatial locality of the data to be preserved in the disk improves access to multidimensional datasets [8].

We think that utilizing the outer zones for improving the sequential access performance and storing files continuously are important for improving the Hadoop application performance.

### 4. Data Transfer Rate of HDD

It is widely recognized that the data transfer rate of HDD for sequential accesses depends on the accessed zone. Higher sequential speed can be obtained in outer zones because those zones can provide more data in a time of a 360-degree rotation than inner zones. In this section, we evaluate sequential read/write access throughput of each zone of the HDD used in the experiments in this paper.

#### 4.1 Experimental Setup

We issued 64 MB read/write commands to the 500 GB HDD for Hadoop data, which is described in Sect. 3.1, and measured sequential access throughput of each zone. We issued command to HDD without filesystems and with Ext2, Ext3, and Ext4. The other experimental setups are the same as that of Sect. 3.1.

#### 4.2 Experimental Results

The experimental results are shown in Fig. 12 to Fig. 15.
horizontal axis in each graph means address in HDD. The vertical axis in each graph indicates time to complete the 64 MB read/write. Therefore, the smaller the time is, the better the performance of the zone is.

Figure 12 depicts the measured throughput without filesystem. Figure 13, Fig. 14, and Fig. 15 show the performance with filesystem Ext2, Ext3, and Ext4, respectively. In Fig. 12, the average time to read in the first zone is 0.339 sec, while it is 0.660 sec in the second last zone. In the case of write, those of the first zone and the last zone are 0.343 sec and 0.667 sec, respectively. These indicate that sequential access speed strongly depends on accessed zone. We can see the similar results with filesystem. Figure 13 to 15 also imply that the outer zones, which have lower addresses, provide better performance than inner zones. All those figures demonstrate that sequential access performance strongly depends on zone. The zone with the lowest address processes a request with twice the throughput of the zone with the highest address.

4.3 Discussion

The previous experiment shows that choosing the location to store data has a large effect on performance. The location to store is determined by the filesystem in most operating systems, including Linux. However, the current filesystem does not have information on the access pattern for it. Hence, filesystem cannot optimize layout of files based on access pattern. As a result, we can conclude that the performance of MapReduce jobs which have bottleneck in sequential I/O, such as TeraSort, can be improved by controlling the file storing location by adjusting filesystem behavior.

5. Proposed Method

In this section, we introduce our proposed methods using an existing filesystem for improving sequential I/O speed by statically or dynamically controlling file storing location. In addition, we explain the proposed striping layout for effectively applying the static method for virtualized environment using image files.

5.1 Static Control of File Storing Location

In work [3], [4], we have proposed a method for placing Hadoop temporary intermediate files in outer zones of an HDD, which are high-speed areas, by static filesystem control. Filesystem manages block usage, used or free, using block bitmap table or similar data structure. In the cases of
Ext2, Ext3, and Ext4, it is managed with block bitmap. Our method [3, 4] sets bits other than for lower address as used. With the method, data blocks for intermediate data are allocated only in outer zones. As a result, disk I/O performance in the Reduce node is expected to be significantly improved. In this paper, we call this method “static method” because this controls filesystem block bitmap statically.

In the work [3, 4], we have introduced our implementation of the static method using Ext2 and Ext3, which are popular filesystem implementations. These filesystem implementations manage HDD space with 4 KB blocks. These create block groups that are composed of blocks. Typically, the size of a block group is 128 MB and a block group includes 32,768 blocks. Each block group has its own block bitmap, inode bitmap, and inode table. Block bitmap manages usage of blocks in the block group._inode bitmap and inode table manage usage of inode number and inode information in the block group, respectively. Our implementation sets bits for blocks with higher addresses as used and forbids blocks of inner zone to be used if there are enough unused blocks with lower addresses.

### 5.2 Dynamic Control of File Storing Location

In this subsection, we propose to improve the static method by enabling dynamic control of filesystem block bitmap. The static method does not scale usable area in application runtime. In the case of running TeraSort for 16 GB input data with three HDFS replicas, the Reduce node stores 12 GB input data, 4 GB data for Map, 12 GB intermediate data, and 16 GB files for output in its filesystem. Thus, the node has up to 44 GB of files. Because the static method has to allocate usable areas according to the maximum required size during application execution, the static method allocates 44 GB usable space at least. This method does not adjust the size of usable space to the needed size which changes dynamically. When an application requires little space, the static method provides excess space and a file may be stored in an inner zone, which is slower zone, in the usable space. In addition, files may be placed far from each other.

In order to avoid excess allocation, the dynamic method allocates a usable space as needed at the time. It expands the usable space as the required size increases. In this paper, we call this method “dynamic method.”

The implementation of the dynamic method has two functions, the periodical monitoring and the dynamic expansion of usable space, in addition to the implementation of the static method. Each function has its own thread and works simultaneously with Hadoop jobs.

The monitoring function periodically checks the number of free blocks in filesystem and calculates the number of usable blocks by subtracting the number of blocks disabled by the proposed method from the number of free blocks obtained by filesystem. In the cases of Ext2 and Ext3, the number of free blocks is stored as s_free_blocks_count in the filesystem superblock. This function periodically checks this value.

The dynamic expanding function is invoked when the monitoring function detects that the number of available usable blocks is less than the threshold. When invoked, it repeats enabling blocks in the fastest zone until the number of available usable blocks exceeds the threshold. These functions avoid placing files in an inner zone unnecessarily and effectively utilize the outer zones. In addition, storing files from a lower address area to a higher address area and keeping available space little tends to store files continuously, i.e. avoid fragmentation, and improve sequentiality of file accesses. The comparison between the static and dynamic methods is presented in Sect. 8.

### 5.3 Striping Layout

In this section, we propose a method for improving I/O performance of Hadoop applications in virtualized environment. We assume the number of virtual machines in the physical machine is given in this paper. Figure 16 illustrates the overview of our method, called striping layout. This method splits each virtual machine image file by the constant sizes into several fragmented files. Then, the fragmented files are placed with striped layout. In this situation, the low addresses of all the virtual disks correspond to the low address of the physical disk, and then applications in all the guest operating systems can use the high speed areas of the physical storage device. In addition, physical seek distances between file storing places also can be decreased.

This method is implemented by re-writing block bitmaps in the host operating system filesystem like the static control of file storing location described in Sect. 5.1. On creating virtual machine image files, this method disables the areas except for striping layout. This method creates image files using not copy-on-write type but raw disk type. For applying this method, intermediate files have to be created in the disk volume that is managed by the proposed method. It is achieved by configuring the location of the files in the setting of core-site.xml or mounting the volume as the directory of intermediate files in the guest operating system.

### 5.4 Implementation

In this subsection, we explain the implementation of our methods. The Ext2 and Ext3 filesystems manage block
usages, used or free with block bitmap. The block usage information can be controlled by opening the special file of the block device of the disk and modifying the flags in the block bitmap table in the filesystem with the administrative authority of the operating system.

Our methods can obtain the location of the block bitmap table according to the specification of these filesystems as follows. The detailed information of the disk layout of these filesystems is described in Appendix A. These methods can get various parameters of the filesystem, e.g., the sizes of the block group and inode, by reading the first block in the filesystem, which is called superblock. The first block of each block group, which stays from 0 byte to 4095 bytes in the group, contains the superblock. The second block, which stays from 4096 bytes, contains the group descriptor tables. The size of each group descriptor table is 32 bytes. For example, the group descriptor table of the block group 0 is between 4096 and 4127 bytes. The first four bytes of each group descriptor table indicates the starting block address of its block bitmap. The next four bytes shows the address of the inode bitmap. The next four bytes does the address of the inode table. Our methods obtain the address of the block bitmap table and seek to the address. These then modify the bitmap in order to control the file placing location.

The dynamic method obtains the number of the available blocks from the filesystem. Our method does not modify this information. Thus, the method can get the actual number.

Our methods do not require to modify the implementation of Hadoop and applications. The methods can be applied to any version of Apache Hadoop straightforwardly. In addition, the existing filesystems, such as Ext2, can be used with our methods. Therefore, we can say that our methods can be applied easily and independent to applications. As described, our methods require the administrative authority for modifying the block bitmap in the filesystem. This is a precondition of our methods.

6. Evaluation

In this section, we present performance evaluation of our proposed methods.

6.1 TestDFSIO in Physical Environment

We executed TestDFSIO in physical environment with the normal method, the static method, and the dynamic method. TestDFSIO is an I/O benchmark application included in Hadoop distribution. The normal method represents that without both of the static and dynamic methods. The entire blocks, including blocks in inner zones, in the HDD are used with the normal method. TestDFSIO was performed ten times with each condition. The file size of read or write is 4 GB. Each slave node processes two files. In the cases of the Hadoop cluster has four slave nodes, the total processing size is 32 GB. We conducted the experiments in the environment with HDFS block size 64 MB. The other experimental setup is same as that of Sect. 3.

In this environment, all the blocks in 500 GB HDD are unused before the experiments. For this situation, the static method forbade the blocks with addresses higher than 40 GB to be used. The threshold of the dynamic method for expanding usable area is 5 GB. Monitoring period is 5 sec.

Figure 17 shows the write performance of TestDFSIO of each execution. Figure 18 shows the read performance. The vertical axes mean I/O performance. Figure 17 indicates that there is no difference of the average sequential write performances of each method. Figure 18 implies that the static and dynamic methods increase the average sequential read performance. The figure indicates also that the static method, which does not have the monitoring and dynamic expanding processes, provides slightly higher read performance than the dynamic method.

6.2 TeraSort in Physical Environment

We executed TeraSort in physical environment with the normal method, the static method, and the dynamic method. The normal method represents that without both of the static and dynamic methods. The entire blocks, including blocks in inner zones, in the HDD are used with the normal method. TeraSort was performed ten times with each condition. In the single-Reduce-tasked job, we conducted experiments in the environment of input data size 16 GB with HDFS block...
In this environment, all the blocks in 500 GB HDD are unused before the experiments. For this situation, the static method forbade the blocks with addresses higher than 60 GB to be used. The threshold of the dynamic method for expanding usable area is 5 GB. Monitoring period is 5 sec.

The measured performance in terms of the average time to complete the job, when the input data size is 16 GB and HDFS block size is 64 MB, is depicted in Fig. 19. That with 128 MB HDFS block size is shown in Fig. 20. The vertical axis of each graph implies the average execution time of the job. Figure 21 shows the performance of each execution.

In the figure, the horizontal axis shows the serial number of TeraSort execution. The vertical axis of the graph is the time to complete each TeraSort execution.

All the figures indicate that dynamic method provides the highest performance. From Fig. 19, we can observe 22.3% and 41.2% performance improvement from the normal method though the static and dynamic method, respectively. Figure 20 indicates 24.7% and 28.6% improvement through the static and dynamic method, respectively. From these, we can say that our methods are effective for improving performance independent to HDFS block size.

From Fig. 21, we can see that high performance is not always obtained with the normal method. In contrast, high performance is always obtained with the static and dynamic methods. These are because files are sometimes placed in an inner zone with the normal method and they are always put in the outer zone with the static and dynamic methods. Comparing performances of the static and dynamic methods, we can observe that performance distribution of the dynamic method is smaller and time to complete the job of the dynamic method is always shorter. From these, we can conclude that the dynamic method can improve effectively the static method.

The measured performance of multiple-Reduce-tasked job, when the input data size is 32 GB and HDFS block size is 64 MB, is depicted in Fig. 22. The vertical axis implies the average execution time of the job. All the figures indicate that the dynamic method provides the highest performance. From Fig. 22, we can observe 13.2% average performance improvement from the normal method.

6.3 TestDFSIO in Virtualized Environment

We executed TestDFSIO in virtualized environment with the normal method, the existing method, and the striping method. The normal method represents that without both of the static method and the striping layout. The entire blocks, including blocks in inner zones, in the HDD are used with the normal method. The existing method means combination of the normal layout in the host operating system and the static method in the guest operating system. The striping method means that combination of the striping layout in the
Table 4  The specification of the physical machine.

|                      |                |
|----------------------|----------------|
| OS                   | CentOS 6.5 x86_64 minimal |
| Kernel               | Linux 2.6.32.57   |
| CPU                  | AMD Turion II Neo N54L Dual-core Processor 2.2GHz |
| Memory               | 16 GB            |
| HDD                  | 500 GB x3        |
| Virtualization System| KVM             |

Table 5  The specification of the virtual machine.

|                      |                |
|----------------------|----------------|
| OS                   | CentOS 6.5 x86_64 minimal |
| Kernel               | Linux 2.6.32.57   |
| CPU                  | AMD Turion II Neo N54L Dual-core Processor 2.2GHz |
| Memory               | 2 GB            |
| HDD                  | 64 GB(Ext4), 130 GB(Ext3) |
| Virtualization System| KVM             |

host operating system and the static method in the guest operating system. TestDFSIO was performed five times with each condition. The file size of read or write was 8 GB. Each slave node processed a file. We conducted the experiments in the environment with HDFS block size 64 MB.

The virtualized environment for evaluation is as follows. The system is composed of a physical computer and three virtual machines. These virtual machines are used as slave nodes, and the physical computer is used as a master node. A Hadoop cluster is constructed with this master node and these three slave nodes. The filesystem in the guest operating system is Ext3. The specification of the physical machine and the virtual machines are shown in Table 4 and Table 5, respectively. The other experimental setup is same as that of Sect. 3.

In this environment, all the blocks in 500 GB HDD are unused before the experiments. For this situation, the existing and striping method forbade the blocks with addresses higher than 50 GB in data HDD in the guest operating system to be used.

Figure 23 shows the read performance of TestDFSIO of each execution. Figure 24 shows the write performance. The horizontal axes of these figures mean the sequence number of each execution. The vertical axes mean I/O performance. Figure 23 indicates that the average sequential read performances of the striping and existing method are higher than that of the normal method by 40.9% and 10.1%, respectively. Figure 24 implies that the striping method increases also sequential write performance. However, the improvement is smaller than that of sequential read. This is mainly because delayed writing function in the operating system collected write requests effectively and made storage accesses sequential. I/O schedulers, such as CFQ, usually spools write requests longer time. In contrast, read requests are not spooled long time. Delaying write requests does not decline application performance because the application can proceed after receiving response of the request from the operating system. However, in cases of delayed read, an application cannot continue processing without receiving the requested data. Thus, operating systems delay write requests more actively than read requests.

6.4 TeraSort in Virtualized Environment

We executed TeraSort in virtualized environment with the normal method, the existing method, and the striping method. TeraSort was performed five times with each condition. We conducted experiments in the environment of input data size was 24 GB and HDFS block size was 64 MB. The other experimental setup is same as that of Sect. 6.3.

In this environment, all the blocks in 500 GB HDD are unused before the experiments. For this situation, the existing and striping method forbade the blocks with addresses higher than 50 GB in data HDD in the guest operating system to be used.

Figure 25 shows the experimental result. The figure demonstrates that the average execution time of the striping and existing methods are shorter than that of the normal method by 23.5% and 4.4%, respectively. The figure indicates that the distribution of the time to complete TeraSort with the striping method is small. This is because the method forces the application to use the same location in the physical storage device. In contrast, the distribution
without the striping method is large. The application sometimes uses a faster area, and sometime uses a slower area.

7. Related Work

Ozawa et al. focused on the single Reduce task in the WordCount job in the I/O bound state, and proposed a method for improving its performance by data compression [10]. Their evaluation demonstrated that their method could improve performance. This work is similar to ours in aspect of focusing on disk I/O of the Reduce task. However, their work does not consider HDD’s feature; further, theirs and our way of achieving the objective are completely different. In addition, both of the works are not in exclusive relation. Thus, applying both methods for achieving further improvement is possible.

In work [11], a method for increasing I/O performance in virtualized environment by controlling block address in inode table in filesystem is proposed. However, the work aimed only to improve random access performance and does not mention sequential I/O throughput, HDD zones, and Hadoop performance. Objectives of both the works are quite different.

Dremel [12], Impala [13], and Camdoo [14] proposed effective methods of aggregation of queries in large-scale data processing. However, these do not discuss HDD zone and sequential I/O throughput. Thus, their approaches are quite different from our method.

Studies on I/O schedulers have been published for improving I/O performance on HDD [16], [17] and other devices [18]. These studies provide methods for avoiding disturbing sequential accesses. However, these studies do not present discussion on the increase of sequential I/O throughput because determining the file layout is performed by filesystem.

In works [2], [5], the static method for improving the performance of Hadoop MapReduce jobs with single Reduce task was proposed. However, these works do not focus on dynamic optimization of usable space. Thus, performance improvements presented by these works were limited.

In work [3], the dynamic method was proposed. However, the method was not evaluated enough. The work presented evaluation only with a single-Reduce-tasked job. In contrast, this paper gives complement evaluation with various workloads, such as a multi-Reduce-tasked job and large I/O benchmark. In addition, application to virtualized environment was not discussed. In work [19], an effective method for applying to virtualized environment was proposed. However, the work provided only discussion on application for virtualized environment.

In work [3], the discussion on the application of the dynamic method for OLAP was presented and the work demonstrated that the method was effective also for OLAP. The objectives of the work [3] and this paper are different.

Wang et al. proposed a method for effectively utilizing the outer zones of ZBR HDD for log-structured filesystem (LFS) [6]. That is a performance-oriented data reorganizing scheme, called PROFS, which improves the I/O performance of LFS. However, the method is specialized for the LFS and its garbage collection. Its evaluation was based only on simulation. Thus, a method that can be applied for a usual filesystem, non-LFS, and evaluation with such filesystems are desired. Schindler et al. proposed track-aligned extents, called traxtents, which utilized disk-specific knowledge to match access patterns to the strengths of modern disks [7]. Their method reduced rotational latency and track crossing overheads and could increase disk access efficiency by up to 50% for mid-sized requests. However, the method mainly focused on seeking time and presented little discussion on placing large files in the faster zones in the HDD. In some cases of accessing a large file, their method run slower than the existing methods. Schlosser et al. discussed methods for improving access to multidimensional datasets by placing the data in the disk according to the spatial locality of the data [8]. The work presented a deep discussion also on mapping between the logical address and the physical place. This work also did not present a deep discussion on improving sequential access performance by placing files in the outer zones. In addition, all of these methods do not consider virtualized environment and not provide any solution for the environment.

Modern filesystems have sophisticated block allocation mechanisms [20], [21]. An allocator reserves a range of blocks for a new file with the reservation based block allocation. Unlike preallocation, the blocks are only reserved in memory. With delayed allocation, the allocation of new blocks in the filesystem to disk blocks is deferred until writeback time. These attempts reduce fragmentations in the filesystem by improving chances of creating contiguous blocks on disk [20]. Ext4 also has refined allocation policies. First is the multi-block allocator. When a file is first created, the block allocator speculatively allocates 8KB on disk space to the file. The second is delayed allocation. The third is that it tries to keep a file’s data block in the same block group as its inode. The fourth is that all the inodes in a directory are placed in the same block group as the directory. The fifth is that the inode allocator scans the block groups and puts a new directory into the least heavily loaded block groups at creating a directory in the root.
directory. These policies improve continuities of files. For creating non-large persistent files, these should work effectively. However, these do not attempt to place files in outer zones actively. The fifth policy may prevent from placing a large file in an outer zone. Therefore, these policies do not work well for large temporal intermediate files. That is, these policies of these modern filesystems are more suitable for general purposes than the proposed method. The proposed method is more suitable for applications with large I/Os such as Hadoop applications.

8. Discussion

For using the dynamic method, periodical monitoring and dynamic expansion are required. We present discussion on the load of these processes. The monitoring function periodically checks the information about the number of free blocks which is stored in the super block of the filesystem. In environment wherein files are frequently created and deleted, the block for this information is always kept in the page cache. Therefore, monitoring this information does not increase I/O load. In the experiment in Sect. 6, monitoring was executed every 5 sec, and CPU utilization of the process of the monitoring function was 0.03%. From these, we can expect that effect of monitoring on performance of an I/O intensive application is very limited. The dynamic expansion function issues write requests to the blocks of block bitmaps. Because of delayed write in page cache, an access to the physical HDD is not issued immediately. In addition, frequent file creations and deletions re-write the data in the block. Hence, the write requests by the dynamic expansion function do not practically reduce performance of I/O intensive applications.

The monitoring period and the threshold for dynamic expansion are tuning parameters of the dynamic method. An expansion is not invoked between observations. If all the usable blocks are consumed in the monitoring period, no block is available in spite of existence of unused blocks. However, this issue can be avoided by the following way. For simple discussion, we first imagine a system without write cache. With such a system, usable area can be consumed up to \( WrSp \times MoInt \) in the period. \( WrSp \) is the writing speed for the HDD and \( MoInt \) is the monitoring interval. Thus, the issue can be avoid by setting the threshold greater than \( WrSp \times MoInt \). With a system with write cache, i.e. a usual operating system, an area up to write cache size can be consumed in a short time because the writing speed for the write cache in the main memory is significantly high. Therefore, the issue can be avoided by satisfying

\[
WrSp \times MoInt + WrCache > ExpTh
\]

(1)

where \( WrCache \) is the write cache size and \( ExpTh \) is the threshold for expansion. The experiments in Sect. 6 met this condition. The write cache size is 2 GB. The maximum writing speed for the HDD is about 200 MB/sec. Monitoring interval is 5 sec. The threshold for expansion was 5 GB. Naturally, tuning these parameters using knowledge on behavior of an application enables further performance improvement.

For implementing our methods, controlling file placing location is required. In this paper, we introduced the implementation using Ext2 and Ext3, with which the control can be achieved by modifying their block bitmap. Implementing using Ext4 also is achieved by controlling location of extents in addition to blocks. Similarly, implementation using other filesystems also is available if control of location is obtained.

Our methods allocate blocks from the outmost zone in the order of arrival of data. We discuss its positive and negative aspects. In cases of not all of the disk blocks are used, using from the outmost zone naturally has a positive effect comparing usage of the entire disk including the inner zones. In case of all the blocks in the disk is used, the used areas with and without the proposed method are the same. However, our method has a positive influence from the time of start of usage to the time when the disk is fulfilled. After the disk become fully is used, the method is neutral. If the files in outer and inner zones are accessed more frequently, the method increases and decreases the performance, respectively. Further performance improvement can be achieved by dynamic relocation, which is moving frequently accessed files to the outer zones.

We compare the static and dynamic methods. The most important benefit of the dynamic method is its effective usage of the outer zones. In a case of the size of the storage area required by an application changes dynamically, the dynamic method can utilize the outer zones more effectively. For example, an application requires \( S_{small} \) GB storage area in its first phase, and it does \( S_{large} \) GB at most during its execution, where \( S_{small} \) is smaller than \( S_{large} \). Because the static method does not change the size of the available storage area during application runtime, the static method has to allocate the usable area that can meet the maximum required size, which is \( S_{large} \). In this case, the system provides an exceeding usable area whose size is \( S_{large} \) in the first phase in spite of only \( S_{small} \) GB is required. The filesystem places files in this available area. As a result, the filesystem may select a zone that is not the outmost. On the other hand, the dynamic method provides only the required size at each time. Therefore, files are forced to be placed in the outmost zone. The benefit of the static method is its simpleness. The static method does not require the monitoring and expanding processes. It does not require a discussion on their parameters. Thus, the static method may outperform the dynamic method in cases of the size of the required disk area is given and does not change during its execution.

Next, we consider the assumption on disk drives. Our methods assume that the higher sequential access performance can be obtained with the zones with low addresses. It is widely recognized that many hard disk drives meet this condition. The study [8] deeply investigated the mapping between the logical address and the physical location on the platters. The study showed that some disks did not completely satisfy this condition. However, the zones with low addresses have the higher sequential access performance.
globally even in such cases. The layout mappings that are introduced in the work [8] are described in Appendix B.

Finally, we present an argument about that the proposed methods can coexist with the Hadoop’s volume-choosing policy of DataNode [22]. Our methods report the false flags on the block usage. However, these methods provide true information on the number of available blocks. The number of usable blocks increases up to the actual number automatically with the dynamic method as the number of used blocks increases. The administrator can increase the number manually up to the actual number with the static method.

9. Conclusion

In this work, we focused on I/O bounded Reduce phase in MapReduce jobs, and discussed methods for improving the performance of such jobs using TestDFSIO and TeraSort. We investigated the bottleneck of the job and revealed that the Reduce tasks were bottleneck tasks. We then showed that the tasks were in the I/O bound state and disk I/Os were mainly executed sequentially. In addition, we presented sequential I/O throughput evaluation and demonstrated that the outer zones had higher sequential access speeds. Based on these discussions, we introduced the static and the dynamic methods for improving the sequential I/O access performance by controlling the location to store the intermediate data. In addition, we explained a striping method for a virtualized environment. Our evaluation has demonstrated that our methods have been able to improve the performance of MapReduce jobs.

For future work, we plan to discuss approaches in operating system level, such as implementing a filesystem suitable for large scale data processing.

Acknowledgments

This work was supported by CREST, JST.

This work was supported by JSPS KAKENHI Grant Numbers 24300034, 25280022, 26730040, 15H02696.

References

[1] Apache Hadoop, available from <http://wiki.apache.org/hadoop> (accessed 2015-07-01)
[2] E. Fujishima and S. Yamaguchi, “Improving the I/O Performance in the Reduce Phase of Hadoop,” The Third International Symposium on Computing and Networking (CANDAR2015), pp.82–88, 2015.
[3] E. Fujishima and S. Yamaguchi, “Dynamic File Placing Control for Improving the I/O Performance in the Reduce Phase of Hadoop,” Proc. 10th International Conference on Ubiquitous Information Management and Communication (IMCOM ’16). ACM, Article 48, 7 pages, 2016.
[4] K. Nakashima, E. Fujishima, and S. Yamaguchi, “File Placing Control for Improving the I/O Performance of Hadoop in Virtualized Environment,” The Fourth International Symposium on Computing and Networking (CANDAR2016), pp.402–407, 2016.
[5] E. Fujishima and S. Yamaguchi, “I/O Performance Improvement of the Reduce Phase of Hadoop,” The Fourteenth Forum on Information Technology (FIT2015). RC-003, 2015. (in Japanese)
[6] J. Wang and Y. Hu, “PROFS-performance-oriented data reorganization for log-structured file system on multi-zone disks,” MASCOTS 2001, Proc. Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Cincinnati, OH, pp.285–292, 2001.
[7] J. Schindler, J.L. Griffin, C.R. Lumb, G.R. Ganger, “Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics,” Conference on File and Storage Technologies 2002 (FAST 2002), pp.259–274, 2002.
[8] S.W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G.R. Ganger, “On multidimensional data and modern disks,” Conference on File and Storage Technologies 2005 (FAST 2005), pp.225–238, 2005.
[9] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” Proc. OSDI’04: Sixth Symposium on Operating System Design and Implementation, vol.6, USENIX Association, Berkeley, CA, USA, p.10, 2004.
[10] T. Ozawa, M. Onizuka, Y. Fukumoto, and S. Moriai, “MapReduce optimization using mapper-side aggregation,” IPSJ Journal: Advanced Computer Systems, vol.6, no.3, pp.71–81, Oct. 2013. (in Japanese)
[11] M. Yamada and S. Yamaguchi, “FileSync System Layout Reorganization in Virtualized Environment,” The 9th IEEE International Conference on Autonomic and Trusted Computing (IEEE ATC 2012), ATC4-2, 2012.
[12] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakakis, “Dremel: interactive analysis of web-scale datasets,” Proc. VLDB Endow., vol.3, no.1-2, pp.330–339, 2010.
[13] Cloudera Impala, available from <http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html>
[14] P. Costa, A. Donnelly, A. Rowstron, and G. O’Shea, “Camdoo: exploiting in-network aggregation for big data application,” Proc. 9th USENIX conference on Networked Systems Design and Implementation (NSDI’12), p.3, 2012.
[15] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, “A platform for scalable one-pass analytics using MapReduce,” Proc. 2011 ACM SIGMOD International Conference on Management of data (SIGMOD ’11), pp.985–996, 2011.
[16] J. Axboe, “Linux block io - present and future,” Proc. Ottawa Linux Symposium, pp.51–61, Ottawa Linux Symposium, 2004.
[17] S. Iyer and P. Druschel, “Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O,” SIGOPS Oper. Syst. Rev. 35, 5, pp.117–130, Oct. 2001.
[18] Y. Nakamura, S. Nomura, K. Nagata, and S. Yamaguchi, “[I/O Scheduling in Android Devices with Flash Storage],” 8th International Conference on Ubiquitous Information Management and Communication ACM IMCOM (ICUIMC), Article 83, 7 pages, 2014.
[19] K. Nakashima, E. Fujishima, and S. Yamaguchi, “Hadoop I/O Performance Improvement in Virtualized Environment by Non-continuous VM Image File Layout in Host Operating System,” SIG Technical Reports, vol.2016-OS-137, no.4, pp.1–6, 2016. (in Japanese)
[20] M. Cao, T.Y. Ts’o, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas, “State of the Art: Where we are with the Ext3 filesystem,” Ottawa Linux Symposium 2005 (OLS 2005), pp.69–95, 2005.
[21] djwong, “Block and Inode Allocation Policy,” https://ext4.wiki.kernel.org/index.php/Ext4_Disk_LAYOUT#Block_and_Inode_Allocati on_Policy accessed Sept. 1. 2017.
[22] L. Xu, “How-to: Use the New HDFS Intra-DataNode DiskBalancer in Apache Hadoop,” Cloudera Engineering Blog, https://blog.cloudera.com/blog/2016/10/how-to-use-the-new-hdfs-intra-datainode-disk-balancer-in-apache-hadoop/ accessed Sept. 1. 2017.
Appendix A: Disk Layout

Figure A-1 illustrates the disk layout of the Ext2 and Ext3 filesystems. The entire HDD is divided into partitions. A filesystem is constructed in each partition. A filesystem of Ext2 and Ext3 is divided into block groups whose sizes are typically 128 MB. Each block group has a superblock, descriptor table, block bitmap, inode tables, and data blocks. When the filesystem allocates a new block, it checks the block bit of each block. If the block bit of a block is 1, the block is not allocated as a new block.

Appendix B: Layout Mapping

Three layout mappings between the logical address and the physical locations in the platters in disk were introduced in the work of [8]. Figure A-2 illustrates these mappings. In the cases of (A) and (B), the higher addresses always stay in the outer zones. In the case of (C), we can see that the arrow points right in several parts, for example the right bottom part. However, we can see that the logical address and the physical place, outer or inner, has a strong relationship even in this case.

Eita Fujishima received the B.E. degrees in Engineering Kogakuin University in 2015. He now stays in Kogakuin University to study electrical engineering and electronics.

Kenji Nakashima received the B.E. degrees in Engineering Kogakuin University in 2016. He now stays in Kogakuin University to study electrical engineering and electronics.

Saneyasu Yamaguchi received Engineering Doctor’s degree (Ph.D.) at Tokyo University in 2002. During 2002-2006, he stayed in Institute of Industrial Science, the University of Tokyo to study I/O processing. He now with Kogakuin University. Currently his researches focus on operating systems, virtualized systems, and storage system.