Hindawi
Journal of Optimization
Volume 2018, Article ID 2068490, 13 pages
https://doi.org/10.1155/2018/2068490

Research Article

Topology-Aware Strategy for MPI-IO Operations in Clusters

Weifeng Liu,[1] **Jie Zhou**,[2] **and Meng Guo**[3]

[1]*Institute of Applied Physics and Computational Mathematics, Beijing 100088, China*
[2]*State Grid Shandong Electric Power Company, Information and Communication Company, China*
[3]*Shandong Computer Science Center (National Supercomputer Center in Jinan), Shandong Provincial Key Laboratory of Computer Networks, Qilu University of Technology (Shandong Academy of Sciences), China*

Correspondence should be addressed to Meng Guo; guomeng@sdas.org

Received 2 March 2018; Revised 30 August 2018; Accepted 16 October 2018; Published 19 November 2018

Academic Editor: Wlodzimierz Ogryczak

This paper presents the topology-aware two-phase I/O (TATP), which optimizes the most popular collective MPI-IO implementation of ROMIO. In order to improve the hop-bytes metric during the file access, topology-aware two-phase I/O employs the Linear Assignment Problem (LAP) for finding an optimal assignment of file domain to aggregators, an aspect which is not considered in most two-phase I/O implementations. The distribution is based on the local data stored by each process, and its main purpose is to reduce the total hop-bytes of the I/O collective operation. Therefore, the global execution time can be improved. In most of the considered scenarios, topology-aware two-phase I/O obtains important improvements when compared with the original two-phase I/O implementations.

1. Introduction

A large class of scientific applications access a high volume of data frequently during their execution. Scalable solutions for efficient and concurrent access to storage are offered by parallel file systems such as Lustre, PVFS, and GPFS. The scientific applications access these parallel file systems through interfaces such as POSIX and MPI-IO [1] or high-level libraries which are based on MPI-IO. In this paper we target optimizing the implementation of MPI-IO interface inside ROMIO, which is the most popular MPI-IO distribution.

Most parallel applications do the computation and I/O alternatively. During the I/O phase, each process often issues a large amount of small noncontiguous I/O requests to access a common data set. These requests usually cause severe overall I/O performance degradation. In order to optimize the performance of the I/O system, the two-phase I/O algorithm is used to merge small individual requests into larger continuous requests. In this work we focus on improving two-phase I/O technique. We have designed and evaluated the topology-aware two-phase I/O technique in which file data access is not only dependent on the data distribution of each process but also dependent on the mapping of processes to computing resources. The comparison with other version of two-phase I/O shows that an important reduction of the run time can be obtained through our technique.

Cluster systems now are moving towards exascale with the high performance interconnection network and many-core architectures. Such systems are getting more and more hierarchical in their interconnection network and node architecture. Processes have different performance levels when communicating at various hierarchies. It is therefore critical for the MPI-IO libraries to reasonably handle the communication demands during the I/O procedure of high performance computing (HPC) applications on such hierarchical systems. MPI-IO is the predominant I/O standard for HPC applications in clusters. During the collective I/O procedure defined in MPI-IO, multiple aggregators exchange data with specific processes. However current MPI-IO optimization strategies do not take the communication pattern and network topology into consideration. In this work, we have designed the topology-aware two-phase I/O, which can improve the shuffle phase of collective I/O operations by carefully placing the aggregators on proper nodes. We have integrated the node physical architecture with network

topology and used graph theory inside MPI-IO library to override the current trivial implementation.

On massively parallel clusters, parallel jobs typically acquire a fraction of the available nodes, which are discontinuous and do not correspond to any regular topology, even when the cluster does. On the other hand for modern machines, contention on specific links limits the communication performance. By suitably assigning processes on proper nodes of clusters, substantial communication and performance improvements on large parallel machines can be achieved. Recently the hop-bytes metric [2, 3], defined as the sum over all the messages of the product of number of hops the message has to traverse and the message size, has attracted much attention. For cluster, this equals the total communication volume. The reason of using the hop-bytes metric is that if the total communication volume is high, then the contention for specific links is also much more likely to increase, and the links would then become communication bottlenecks. During MPI-IO procedure, selection of aggregators and assignment of file domains that taking hop-bytes into account can significantly reduce communication overhead. Although the communication bottleneck caused by link contention is not directly measured by this metric, low values of this metric mean smaller communication overheads. When using this metric, we only have to measure the machine topology; the routing information is not necessary.

This paper is structured as follows. Section 2 introduces the related work. The implementation detail of two-phase I/O is described in Section 3. Section 4 gives the description of the topology-aware two-phase I/O. Section 5 overviews the evaluated application, in addition to the evaluation results that compare the topology-aware two-phase I/O with the original version of two-phase I/O.

2. Related Work

Due to the increasing requirements of applications for data movement to memory or storage, parallel I/O is an active research topic now. From the perspective of file system, highly scalable parallel file systems such as GPFS [4] or Lustre [5] are widely used. At the application level, parallel I/O libraries MPI-IO, which is part of the MPI-2 standard, is commonly deployed. With MPI-IO, collective I/O allows achieving improved performance. Various collective I/O write algorithms are evaluated by Chaarawi et al. [6]. Some researches try to optimize collective I/O with techniques such as automatic collective I/O tuning with machine learning [7] and process placement based on the I/O pattern [8]. Two-phase I/O is the de facto collective I/O algorithm [9]. It adds a shuffle phase in collective I/O phases by aggregating data on a subset of processes (aggregators) before writing it onto the parallel file system. ROMIO [10] is a popular implementation of MPI I/O using two-phase I/O and it has been included in MPICH, Open MPI, IBM MPI, NEC MPI, SGI MPI, and HP MPI. Some researches try to improve the two-phase I/O algorithm [11]. Approaches based on double buffers using multithreading to overlap shuffle phase and I/O phase have been studied in [12, 13]. Properly setting the buffer size and the aggregator number is still an important topic [14]. Finally, placing the aggregators on proper cores is a well-known problem. Certain approaches focus on discovering data locality and using a polynomial-time file domain to aggregator assignment algorithm to minimize communication between computing processes and aggregators [15]. Other researchers try to take the routing mechanism into consideration when issuing sparse data access on BG/Q [16]. Previous IBM supercomputers BG/P adopt a general method designed to increase the I/O bandwidth of collective I/O [17]. Tessier et al. [18] use a different approach which combines an optimized buffering system and a topology-aware aggregators mapping strategy targeting any kind of architecture and being extensible to address new tiers of storage. However their approach does not take link contention into consideration and also does not try to minimize the execution time of all aggregators.

3. Internal Structure of Two-Phase I/O

After reviewing the related work on MPI-IO optimization, we introduce two-phase I/O in detail. As the name indicates, two-phase collective I/O consists of an I/O phase and a shuffle phase. In the I/O phase, contiguous data block transfers are performed from or to the parallel file system. In the shuffle phase, by interprocess communication, small file requests of different processes are grouped in larger ones. Before the two phases, the file region which is contained between the minimum and maximum offsets of all file requests is divided into a configurable number of file domains (FD), and each FD is assigned to a chosen process which is called aggregator. All the data that locates inside an FD is aggregated by the related aggregator which is responsible for transferring the FD from or to the parallel file system.

In summary, the procedure of two-phase I/O can be divided into the following stages. Offsets and lengths calculation stage (st1): In this stage each process calculates the offsets and lengths of its access requests and communicates its start and end offsets to other processes. In the end of this stage, all processes have global file access information and the involved file interval can be calculated. File domain assignment stage (st2): In this stage the involved file interval is divided into file domains (FDs) among aggregators. In this way, each aggregator only accesses the data associated with its FD in the following stages. Access request calculation (st3): The portions of the access request of each process are analyzed and which file domains they locate can be calculated. Other processes' requests which lie in the file region of each aggregator are also calculated. Buffer writing (st4): Processes send their data to appropriate aggregators, and the data are stored in the buffer of each aggregator. Disk accessing (st5): For collective write, aggregators collect data from other processes and write them to the file domain; for collective read, aggregators read data from its file domain and send them to other processes. This stage is made as many times as the following calculus indicates: the file domain size of each aggregator divided by size of the collective buffer size.

In the previous implementation of two-phase I-O, the assignment of FD to each aggregator in st2 does not take the network topology and the distribution of data into consideration. Our technique tries to modify the FD assignment

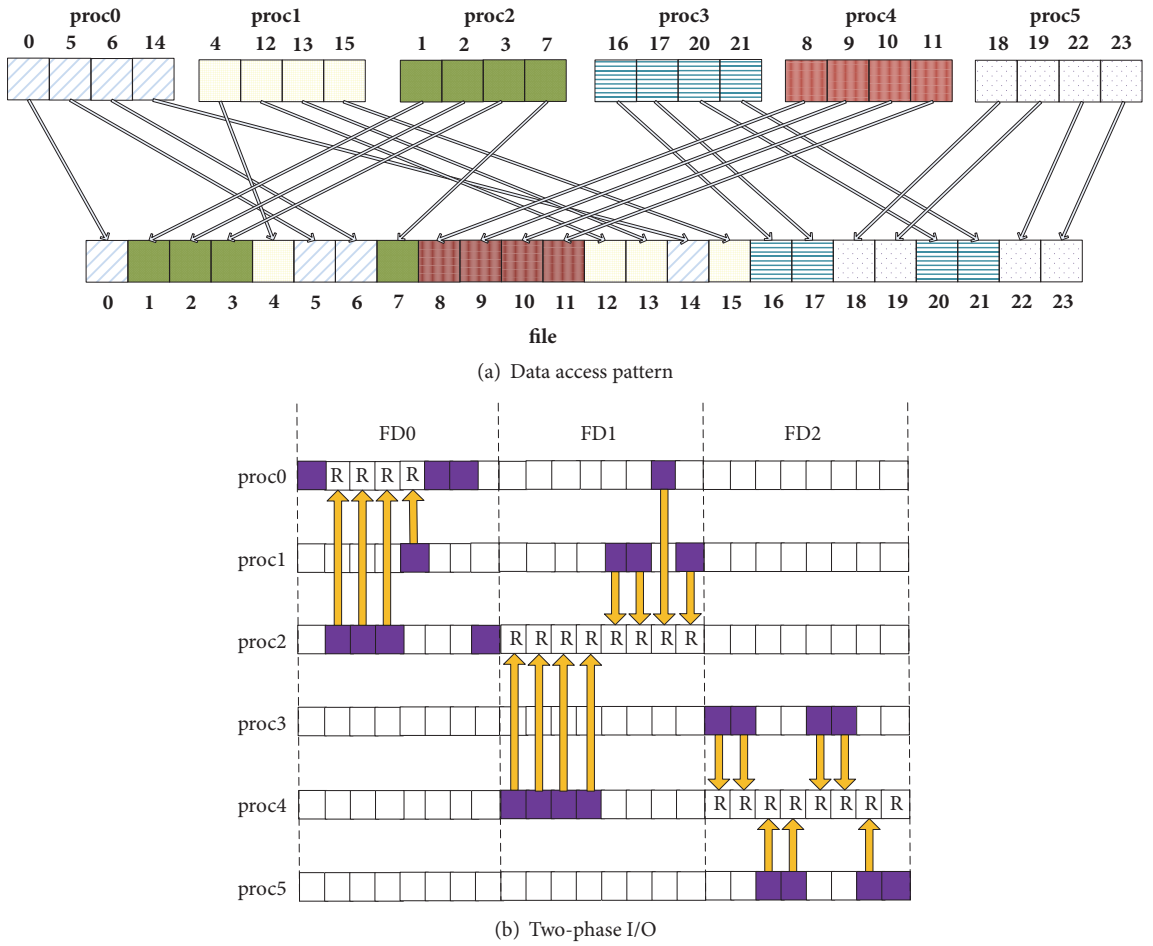


Figure 1: Two-phase I/O. Optimizing collective MPI-IO writes.

strategy, so that the initial distribution of data in the cluster is considered. By means of this strategy it is possible to improve the communication overhead and, therefore, reduce the overall I/O time. Figure 1 details the two-phase I/O technique by an example of 6 processes that write a vector of 24 elements to a file in parallel. In this example, each process writes 4 noncontiguous data blocks. P0, P2, and P4 are chosen as aggregators, and the accessed file region is divided into 3 FDs, which are assigned to them. For this example, in the shuffle phase each aggregator receives 8 data blocks from the target processes and writes them to the file in the I/O phase.

Parallel file systems usually partition a large file into blocks which are striped across many I/O servers. A client should exclusively access a file region to guarantee the cache coherence. A locking mechanism is used by Lustre and GPFS to implement the exclusive access. The lock granularity of these parallel file system is set to their block size. If the involved file region is divided into equal size file domains, a block may expand over two file domains. So when two aggregators simultaneously modify that block, the access requests must be serially served. Liao et al. [19] improved ROMIO and proposed some methods which align each file domain to a lock boundary to prevent lock contention. Some strategies are also designed to avoid lock conflicts in ROMIO's Lustre-specific code. As Figure 2 shows, the minimum and maximum offsets of the accessed file region are aligned to the lock boundaries. The optimization strategy requires that the aggregator number should be a value that can divide the OST number exactly. In Figure 2, the stripes to be read are allocated round robin among aggregators P0 and P1, and a file domain consists of the assigned stripes of an aggregator. The collective buffer size is set to the stripe size. In the I/O phase each aggregator reads a stripe into its collective buffer, and they distribute the cached data to other processes in the shuffle phase.

4. Topology-Aware Strategy for MPI-IO Implementation

In previous implementations, the assignment of the FDs to each process does not take the I/O pattern and network topology into consideration. With the topology-aware two-phase I/O, the assignment of the FDs is dependent on the initial data distribution and the locations of the processes in the cluster. The new strategy tries to minimize the communication workload or latency during the shuffle phase, and it solves the problem based on the Linear Assignment Problem. In this section, we present details about the topology-aware

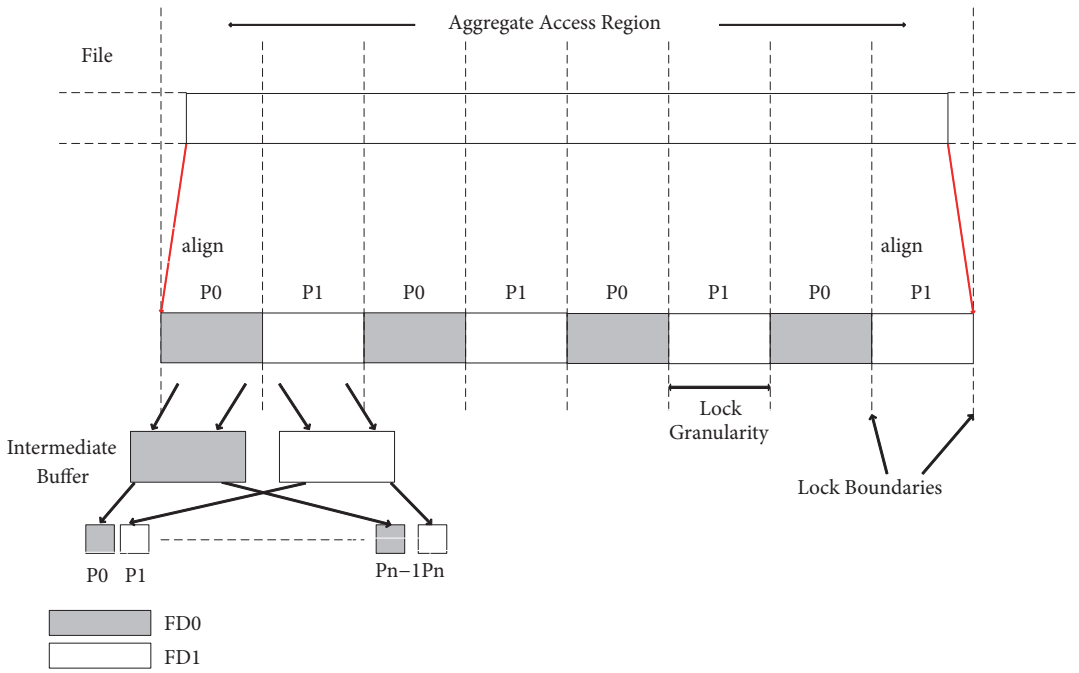


Figure 2: The implementation of two-phase I/O on supercomputers with Lustre installed.

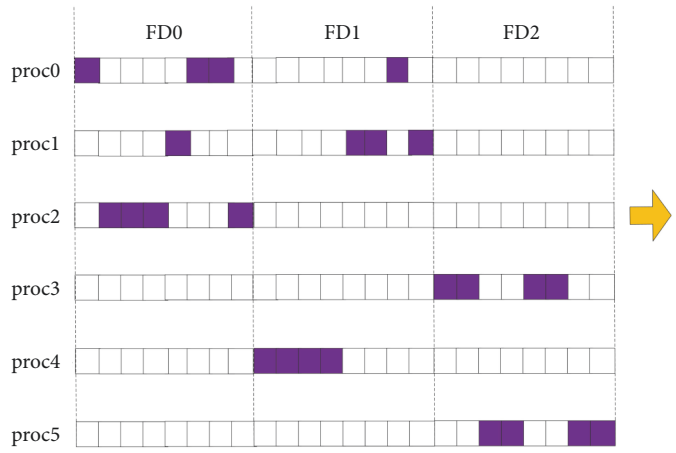
two-phase I/O. At first we show how we get the virtual and the physical topologies. These topologies are constructed as matrices whose elements represent the interprocess communication volume and the network distances between any two cores, respectively.

4.1. Collecting the Applications Communication Pattern Data. To optimize the communication performance, the vital piece of information that we need to get is the target application's communication pattern in the shuffle phase. In this case, the communication pattern is stored in a $p \times a$ matrix (p is the process number and a is the aggregator number) which consists of the volume of data exchanged between each aggregator and their target processes. Fortunately, for two-phase I/O, we can get its communication pattern during the shuffle phase in st3 and we do not need to preliminary run the application.

$$C = \begin{pmatrix} 3 & 1 & 0 \\ 1 & 3 & 0 \\ 4 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix} \quad (1)$$

In order to explain this, we use the example with the same data size and access pattern as shown in Figure 1. The number of intervals in which the file can be divided is set to be equal to the aggregator number, and the FDs can be calculated. The next step is constructing the communication matrix C, which is with so many rows as processes, and so many columns as FDs. Each matrix entry indicates the number of elements of a FD that will be accessed by a process. Matrix C shows the result for our example. Each single FD is assigned to one aggregator, and all processes communicate with the aggregators during the shuffle phase; thus the matrix gives the communication pattern. Figure 3 shows how to calculate the communication matrix. In matrix C, the rows indicate different processes and the columns indicate different aggregators.

4.2. Gathering the Computing Resources Topology. The node architecture and network architecture constitute the computing resource's physical topology. Higher communication performance can be expected between the cores with shorter network distance. The hardware locality (hwloc) library [20] can provide the underlying machine architecture abstraction. It detects the nodes' architectural components such as caches, cores, processor sockets, memory, NUMA, and SMT architecture and represents them as a tree with cores at the leaves and nodes at the top. We need a network discovery module to get a view of the computing resources' physical topology. InfiniBand subnet manager tools such as ibtracert [21] can help us discover the network distance between computing nodes interconnected by InfiniBand network. The module should get the distance information with the tools and merge the result with the nodes' architectural information got by hwloc. The result distance matrix will be used to make FDs assignment. For MPI applications running in a homogeneous cluster, the process with the minimum rank in the communicator will extract the distance between any two nodes using ibtracert. The process then scatters the distance



	FD0	FD1	FD2
P0	3	1	0
P1	1	3	0
P2	4	0	0
P3	0	0	4
P4	0	4	0
P5	0	0	4

Figure 3: The communication between aggregators and their target processes.

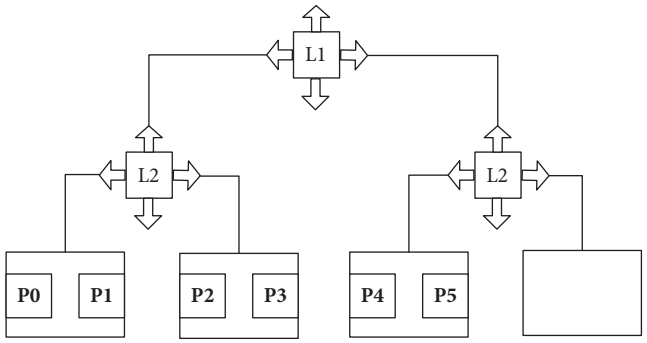


Figure 4: 4 nodes (here, 2-way 1-core) are connected through switches in a network with tree switches.

information to other processes which will integrate it into the node architecture to get the computing resources' full architecture.

$$D = \begin{pmatrix} 0 & 1 & 2 & 2 & 4 & 4 \\ 1 & 0 & 2 & 2 & 4 & 4 \\ 2 & 2 & 0 & 1 & 4 & 4 \\ 2 & 2 & 1 & 0 & 4 & 4 \\ 4 & 4 & 4 & 4 & 0 & 1 \\ 4 & 4 & 4 & 4 & 1 & 0 \end{pmatrix} \quad (2)$$

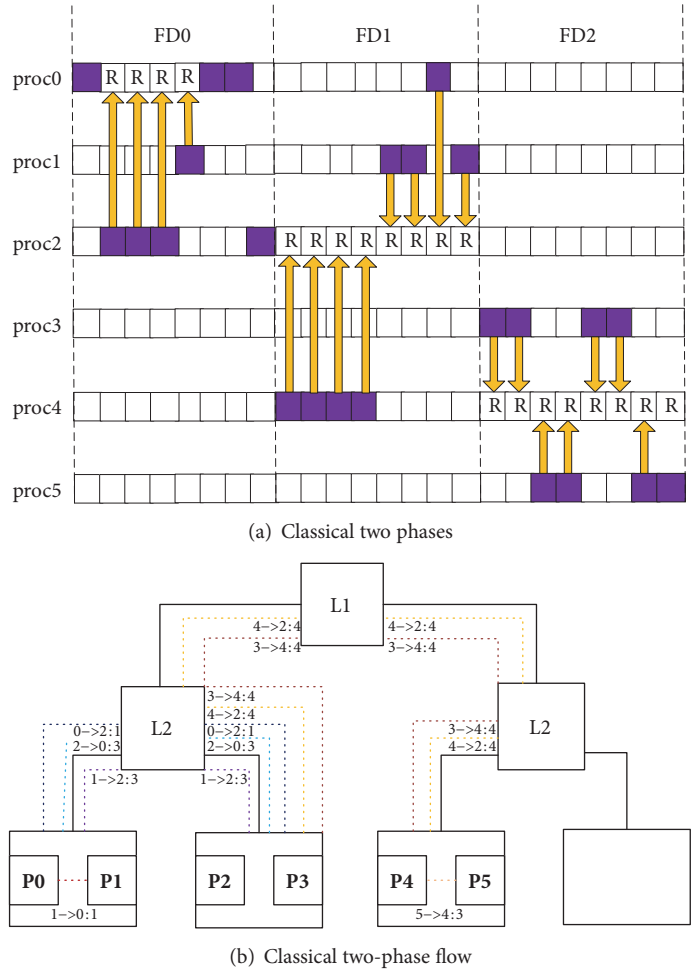
Topology matrix element represents the number of hops (length of the communication path) between two nodes. The farthest nodes get a value of the maximum network hop count for their topology matrix element. The intranode matrix elements are assigned a value that is always smaller than or equal to 1, indicating the fact that intranode communication is faster than internode communication which requires more than 1 network hop. Matrix D shows how the various distance values of Figure 4 are assigned based on the network topology and system architecture. 4 nodes (here, 2-way 1-core) are connected through 3 switches in a network. We assign the minimum distance value for cores on the same node. The next minimum distance value is 2: for instance P0 is able to communicate with P2 with 2 hops. We can assign distance values for other node pairs in the same way.

For classical two-phase I/O, the 3 FDs will be assigned to 3 aggregators running on 3 nodes; in this example they are P0, P2, and P4. The resulting network communication is shown in Figure 5. In this case the maximum link congestion is 15, and the overall I/O performance is affected by the over congested link.

4.3. Topology-Aware Two-Phase I/O. As shown in Figure 5, if the FDs are not properly assigned to the aggregators, the resulting network communication will cause high link congestion; some new mechanisms are needed to reduce the link congestion during the shuffle phase. Our technique is based on minimizing the communication workload and latency. The assignment of the FD to each aggregator in the proposed technique is different from the original version. Now each FD is assigned based on the total hop-bytes of the interprocess communication during the shuffle phase. The number of FDs into which the file can be divided is set by the hints, and the topology matrix D and the communication matrix C have already been calculated out. The next step consists in assigning each interval to each process efficiently.

$$W = C * D = \begin{pmatrix} 9 & 16 & 24 \\ 11 & 17 & 24 \\ 8 & 24 & 20 \\ 12 & 24 & 16 \\ 32 & 16 & 20 \\ 32 & 20 & 16 \end{pmatrix} \quad (3)$$

We use the hop-bytes metric to estimate the communication workload. To minimize the workload, we first calculate the work load matrix W with so many rows as processes and so many columns as FDs. When the jth FD is assigned to process i, entry e_{ij} of the work load matrix indicates the total hop-bytes of the related interprocess communications aiming to collect or scatter the data of the jth FD to or from



(a) Classical two phases

(b) Classical two-phase flow

Figure 5: The classical two-phase I/O and the resulting network communication.

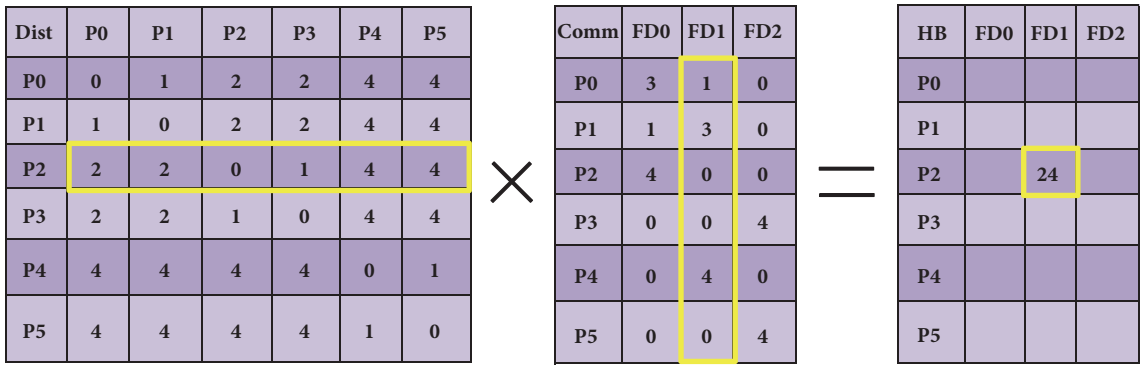
process i; thus e_{ij} is the inner product of the ith row of the topology matrix D and the jth column of the communication matrix C, respectively, so $W = C * D$. Figure 6(a) shows the result for our example. If the second FD is assigned to P2, all processes will communicate with P2 during the shuffle phase to send the data of the second FD stored by them. The message sizes are indicated by the second column of matrix C and the third row of matrix D indicates the distance between the processes. In this example, e_{21} equals 24. We can calculate all the other elements of matrix W in the same way as shown in Figure 6(b).

After getting the workload matrix W, we need to assign all the FDs to proper aggregators. The FD assignment strategy in our work is a Linear Assignment Problem (LAP) which has been well studied in combinatorial optimization and linear programming. LAP is about how to assign n items to n elements given a cost matrix in the best way. In our research, the cost matrix has been got through multiplying two matrices that record the interprocess communication volume and the interprocess distance, respectively. We have to reduce the contention on specific links, and the hop-bytes metric should be minimized. Each of the file domains can be assigned to only one aggregator, and one aggregator can access only one file domain during the I/O phase. In other words, we need to select n elements from the cost matrix, so that in each row and each column there is exactly one selected element, and the sum of them is minimum.

A large number of algorithms have been developed for LAP. The problem of finding the best FDs assignment to some particular processes can be based on the existing solutions of LAP. We have selected for our work the following algorithms, considered to be the most representative ones:

(i) Hungarian algorithm [22]: This is the first polynomial-time primal-dual algorithm solving the assignment problem. It was invented and published by Harold Kuhn in 1955 and has a $O(n^4)$ complexity.

(ii) Jonker and Volgenant algorithm [23]: A shortest augmenting path algorithm was developed to solve the Linear Assignment Problem. This algorithm contains new initialization routines and an implementation of Dijkstra's shortest path method. This algorithm is shown to be uniformly faster than the best algorithms from the literature for both dense and sparse problems computational experiments. It has a $O(n^3)$ complexity.



Dist	P0	P1	P2	P3	P4	P5
P0	0	1	2	2	4	4
P1	1	0	2	2	4	4
P2	2	2	0	1	4	4
P3	2	2	1	0	4	4
P4	4	4	4	4	0	1
P5	4	4	4	4	1	0

×

Comm	FD0	FD1	FD2
P0	3	1	0
P1	1	3	0
P2	4	0	0
P3	0	0	4
P4	0	4	0
P5	0	0	4

=

HB	FD0	FD1	FD2
P0			
P1			
P2		24	
P3			
P4			
P5			

(a) Single hop-byte element calculation

Dist	P0	P1	P2	P3	P4	P5
P0	0	1	2	2	4	4
P1	1	0	2	2	4	4
P2	2	2	0	1	4	4
P3	2	2	1	0	4	4
P4	4	4	4	4	0	1
P5	4	4	4	4	1	0

×

Comm	FD0	FD1	FD2
P0	3	1	0
P1	1	3	0
P2	4	0	0
P3	0	0	4
P4	0	4	0
P5	0	0	4

=

HB	FD0	FD1	FD2
P0	9	16	24
P1	11	17	24
P2	8	24	20
P3	12	24	16
P4	32	16	20
P5	32	20	16

(b) All hop-byte elements calculation

Figure 6: Calculate the hop-bytes when different FDs are assigned to different processes.

(iii) APC and APS algorithms [24]: These algorithms implement the Lawler O(n^3) version of the Hungarian algorithm by Carpenato, Martello, and Toth. APC works on a complete cost matrix, while APS works on a sparse one.

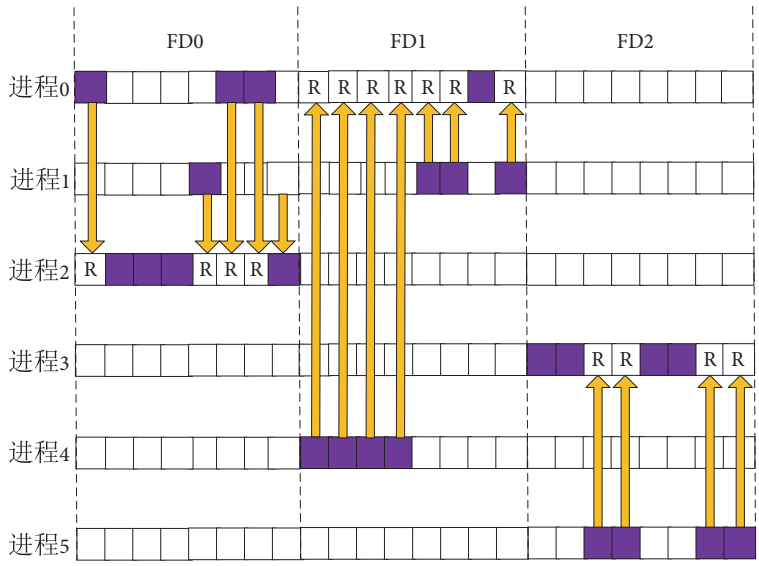
Previous evaluation [15] shows that the fastest algorithm is the Jonker and Volgenant one, and for this reason we have chosen to apply it in our topology-aware two-phase I/O. As shown in Figure 6(b), for our topology-aware two-phase I/O, the three file domains are assigned to P2, P0, and P3, respectively. Each process selected as aggregator writes to file a consecutive data set. As shown in Figure 7, in this case the maximum link congestion decreases to 9, and the overall I/O performance is significantly improved.

5. Performance Evaluation

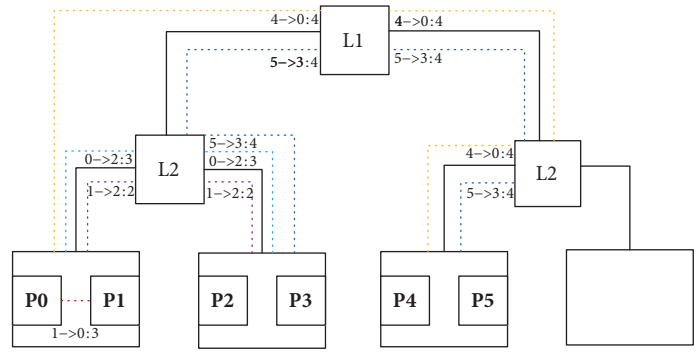
The evaluations in this paper were performed by using two I/O benchmarks. We have compared topology-aware two-phase (TATP) I/O with the original version of two-phase (OTP) I/O implemented in MPICH and locality-aware two-phase (LATP) I/O implemented by Filgueira [15].

5.1. The Experimental Platforms. The tests have been made in our Inspur cluster and Sunway BlueLight running in National Supercomputer Center in Jinan. The Inspur cluster is organized with tree topology; it consists of 4 racks, each of which is composed of 20 Inspur computing nodes interconnected by a 40 Gb/sec InfiniBand switch. All the 4 racks are connected together by a 20 Gb/sec InfiniBand switch. Each computing node runs Red Hat 5.0 with a kernel of 2.6.18 and has two six-core processors and 8 GB memory. The parallel file system installed on the Inspur cluster is Lustre. Sunway BlueLight is organized with fat-tree topology, the water-cooled 9-rack system has 8704 ShenWei SW1600 (16 cores, 140GFLOPS) processors organized as 34 super nodes (each consisting of 256 compute nodes), 150 TB main memory, and 2 PB external storage. The system runs on its own operating system, Sunway RaiseOS, which is based on Linux. The parallel file system installed on Sunway BlueLight is also Lustre. All the computing nodes have the same distance to the I/O server of Lustre file system, so they have almost the same I/O performance in the I/O phase and improving the shuffle phase of two-phase I/O can significantly improve the overall I/O performance.

5.2. The I/O Benchmarks. We have run some benchmarks in our previous work [14], and we test those benchmarks with our new topology-aware two-phase I/O. The first benchmark has collective write/read operations. It reads/writes a three-dimensional double array from/to a file which stores the global array in row-major order. Its access pattern is shown in Figure 8(a). The 2000×2000×2000 double array is chosen and the total file size is about 60GB. Different number of processes are started to run the first parallel I/O benchmark, respectively. The data set is divided into cubes, each of which

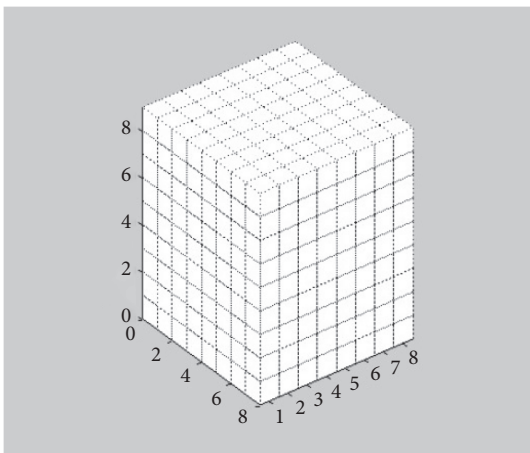


(a) Topology-aware two phases

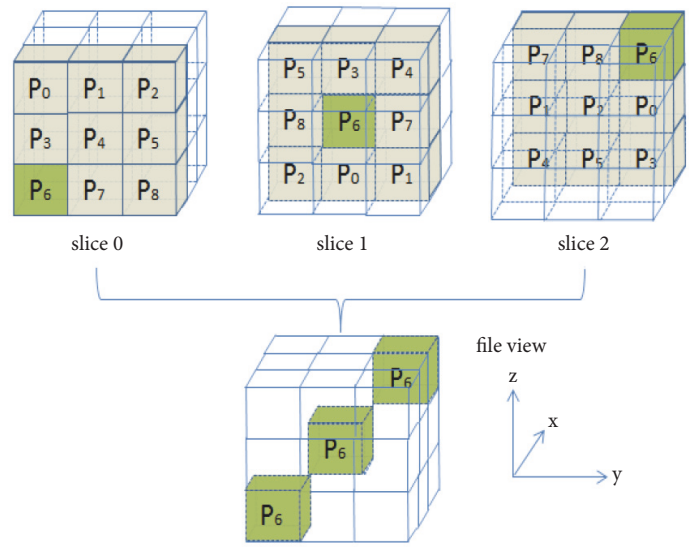


(b) Topology-aware two-phase flow

Figure 7: The topology-aware two-phase I/O and the resulting network communication.

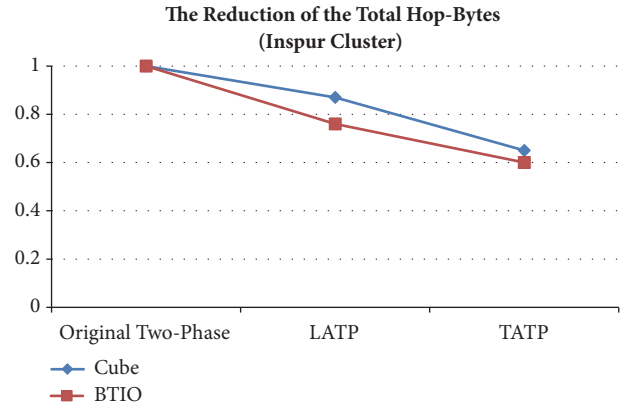


(a) Cube I/O

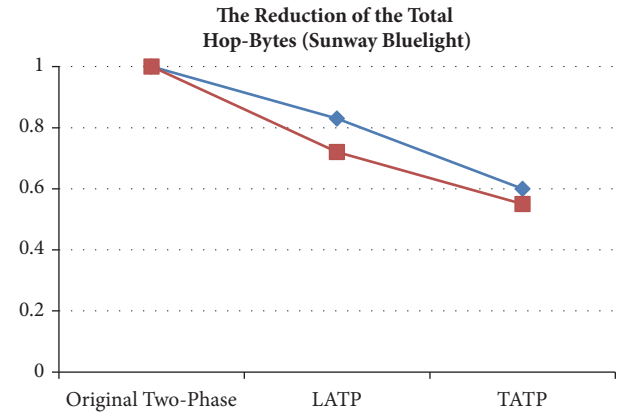


(b) BTIO

Figure 8: The tested benchmarks' access pattern.



(a) The reduction of the total hop-bytes on Inspur cluster



(b) The reduction of the total hop-bytes on Sunway BlueLight

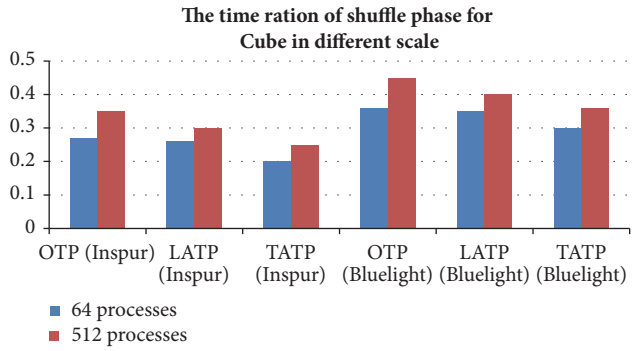
Figure 9: The reduction of the total hop-bytes.

is assigned to one process, and a process needs to access plenty of discontinuous data pieces in the file. File view is created to describe the access pattern and application accesses the file by collective MPI-IO operations. This benchmark represents the I/O patterns of many applications such as volume visualization which displays a 2D projection of a 3D discretely sampled data set.

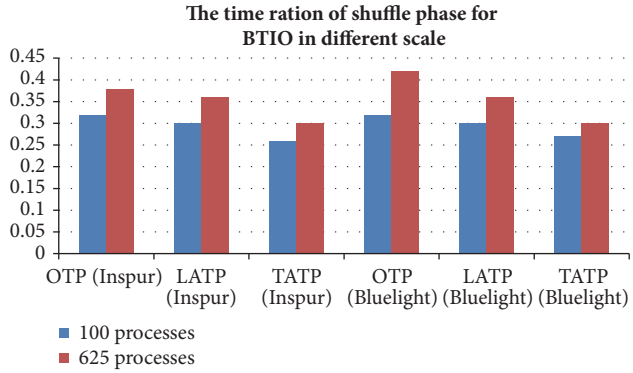
In the second benchmark we implement the file access of BTIO [25] with MPI-IO functions. In this benchmark, a three-dimensional array is partitioned in a block-tridiagonal pattern and assigned across a square number of processes. Each process is responsible for many (the square root of the number of participating processors) subsets of the entire data set. When the process number is nine, Figure 8(b) illustrates how the data set is partitioned. Take process 6 as example, the cube in row 2 and column 0 of slice 0 is assigned to it, in the next slice the cube in row 1 $((2-1) \bmod 3)$ and column 1 $((0+1) \bmod 3)$ is assigned to it, and so on. Every process sets the file view and writes or reads all data subsets with one collective MPI-IO operation. We started different number of processes to run the test, respectively, and set the size of the global double array to 2000×2000×2000.

5.3. Performance Evaluation of Topology-Aware Two-Phase I/O. During the test, the aggregator number and collective buffer size were set to the default value, so on each node the process with the minimum process id will be chosen as aggregator. Firstly we started 512 and 625 processes to run cube and BTIO, respectively. Figure 9 shows the reduction of the total hop-bytes for topology-aware two-phase (TATP) I/O over original two-phase (OTP) I/O and locality-aware two-phase (LATP) I/O for the two benchmarks. We can see that when topology-aware two-phase I/O is applied, the volume of the total hop-bytes is considerably reduced. As we can see in Figures 9(a) and 9(b), for the two benchmarks, the topology-aware two-phase I/O and LATP I/O reduce more hop-bytes on Bluelight than on Inspur cluster. This condition can be explained. The Bluelight has larger scale and runs more jobs than the Inspur cluster. So its jobs usually run on very discrete nodes. With this case, the topology-aware two-phase I/O can obviously reduce the total hop-bytes and improve the collective communication during the I/O procedure.

We have represented the different phases of two-phase I/O as metadata calculation, metadata transformation, data shuffle, and data I/O. We tested the time ratio of data shuffle



(a) The time ratio of shuffle phase when cube runs in different scales



(b) The time ratio of shuffle phase when BTIO runs in different scales

Figure 10: The time ratio of shuffle phase when the two benchmarks run in different scales.

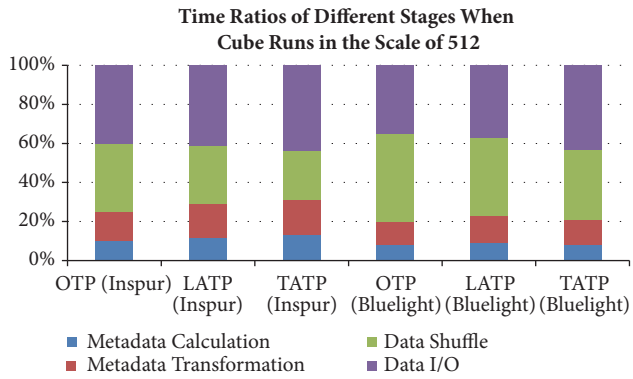
phase when the two benchmarks run in different scale. As we can see in Figures 10(a) and 10(b), the cost of the shuffle stage increases with the process number. Figure 11 represents the time ratios of different two-phase I/O stages when the two benchmarks run in the scale of 512 and 625 processes, respectively. As we can see in Figures 11(a) and 11(b), the slowest stages are data shuffle and data I/O. Based on this, we conclude that the shuffle stage is a bottleneck in two-phase I/O. For this reason, the TATP I/O technique with the aim of reducing the communication cost is necessary. This technique reduces the global hop-bytes of two-phase I/O, so the congestion is reduced and the I/O performance is improved.

We also tested how the process number affects the execution of the benchmarks. 64, 125, or 512 processes were, respectively, started to run the cube, and 100, 625, or 900 processes were, respectively, started to execute BTIO. Figure 12 shows the aggregated I/O speed when different number of processes did the collective I/O operations with different two-phase I/O implementations. We can see that the I/O performance is significantly affected by the process number. As we can see in Figure 12, topology-aware two-phase I/O has the best I/O performance. Note that when using topology-aware two-phase I/O, the total hop-bytes is also the minimum. The I/O speed increases with the decreasing of hop-bytes. These figures show the relevance of the I/O performance and the total hop-bytes. With this technique we can significantly increase the global I/O speed.

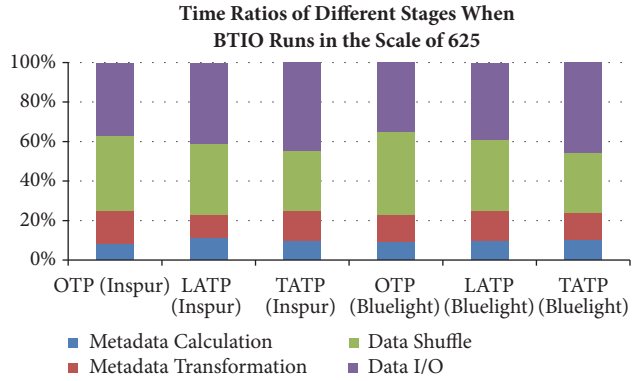
We also tested how the aggregator number affects the benchmarks' execution. With the default configuration, on each node, there is only one aggregator. To do the test, 512 processes were started to run cube and 625 processes were started to run BTIO. During the test, the number of aggregators on each node is set to 1, 2, and 4, respectively, and TATP I/O is used. The aggregated I/O speed is shown in Figure 13(a) and the comparison of the resulting hop-bytes is shown in Figure 13(b). Note that the I/O speed increases with the decreasing of hop-bytes. These figures show the relevance of the aggregator number and the total hop-bytes. Based on the design principle of two-phase I/O, once the aggregator number is set, no matter how we set the collective buffer size, the total hop-bytes during the shuffle phase remain unchanged. So properly setting the aggregator number is important for reducing the total hop-bytes. In our previous work we study how to automatically set the aggregator number and collective buffer size [14]. With this technique we can significantly increase the global I/O speed.

6. Conclusion

In this paper we have presented the topology-aware two-phase I/O (TATP), which optimizes the most popular collective MPI-IO implementation of ROMIO. With the topology-aware two-phase I/O (TATP), the assignment of the FDs depends on the initial data distribution and the locations

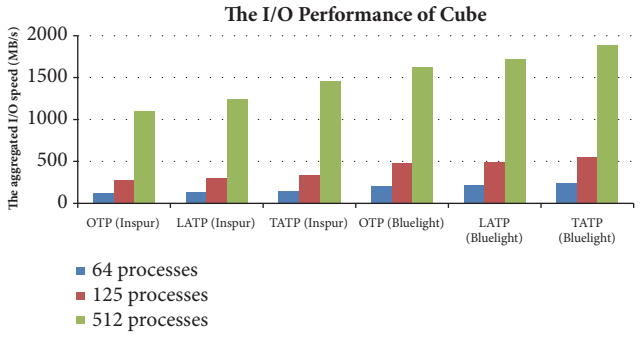


(a) Stages of different two-phase I/O implementations for Cube

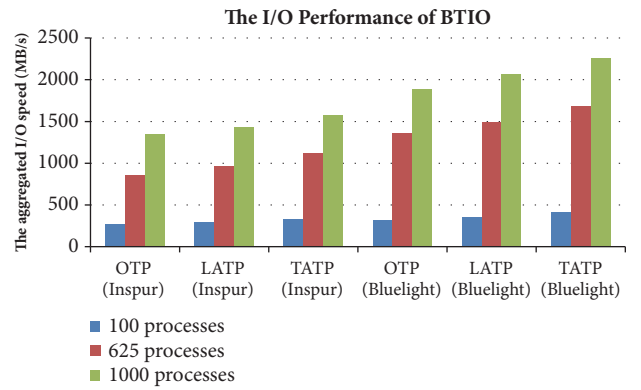


(b) Stages of different two-phase I/O implementations for BTIO

Figure 11: Stages of different two-phase I/O implementations for the two benchmarks.

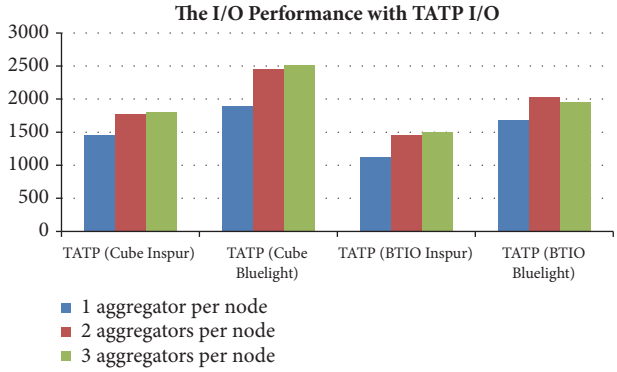


(a) The I/O performance of cube with different two-phase I/O implementation

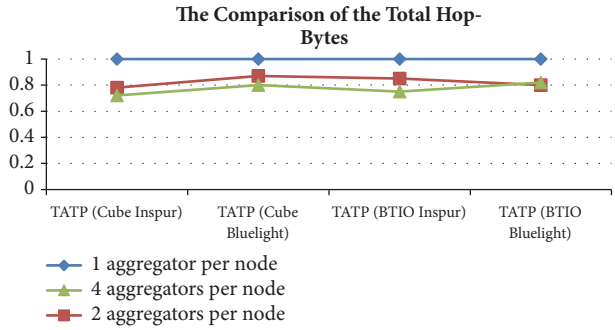


(b) The I/O performance of BTIO with different two-phase I/O implementation

Figure 12: The I/O performance of the two benchmarks with different two-phase I/O implementation.



(a) The I/O performance of the two benchmarks with different aggregator number



(b) The comparison of the total hop-bytes of the two benchmarks with different aggregator number

Figure 13: The relevance of the aggregator number and the total hop-bytes.

of the processes in the cluster. We use the hop-bytes metric to estimate the communication workload. If the total communication workload is low, then the contention for specific links is also much more likely to decrease. The Linear Assignment Problem (LAP) is employed to find an optimal assignment which can minimize the communication workload. As far as we know, this is the first work trying to improve the performance of two-phase I/O through reducing the total hop-bytes. Experiment results show that topology-aware two-phase I/O obtains important improvements when compared with the original two-phase I/O implementations.

Properly setting the aggregator number can significantly reduce the total hop-bytes. In our previous work an auto-tuning framework is used to automatically evaluate different aggregator numbers, respectively, but this approach takes long time to find the best configuration. In the future we will design an algorithm to compute the best configuration directly.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by grants from National Key Research and Development Program (2018YFB0704002) and Key Research and Development Program of Shandong Province of China (2015GGX101028). We thank National Supercomputer Center in Jinan (NSCCJN) and Shandong Province High Performance Computing Center (SDHPCC) for providing experiment environment.

References

[1] R. Buyya, T. Cortes, and H. Jin, "Overview of the mpiio parallel i/o interface," in *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pp. 476–487, IEEE, 1995.

[2] C. D. Sudheer and A. Srinivasan, "Optimization of the hop-byte metric for effective topology aware mapping," in *International Conference on High Performance Computing*, IEEE, 2012.

[3] A. Bhatele, I. Chung, and L. V. Kale, "Automated mapping of structured communication graphs onto mesh interconnects," 2010.

[4] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *FAST '02 Proceedings of the 1st USENIX Conference on File and Storage Technologies*, vol. 2, pp. 19–32, 2002.

[5] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003, 2003.

[6] M. Chaarawi, S. Chandok, and E. Gabriel, "Performance evaluation of collective write algorithms in mpi i/o," in *Proceedings of the International Conference on Computational Science*, pp. 185–194, 2009.

[7] F. Isaila, P. Balaprakash, S. M. Wild et al., "Collective I/O Tuning Using Analytical and Machine Learning Models," in *Proceedings of the Ieee/rsj International Conference on Intelligent Robots and Systems*, vol. 3, pp. 2392–2397, 2015.

[8] V. Venkatesan, R. Anand, J. Subhlok, and E. Gabriel, "Optimized process placement for collective I/O operations," in *Proceedings of the European Mpi Users' Group Meeting*, pp. 31–36, 2013.

[9] J. M. D. Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," *ACM*, 1993.

[10] R. Thakur, W. Gropp, and E. Lusk, "A Case for Using MPI's Derived Datatypes to Improve I/O Performance," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98*, pp. 1–10, 1998.

[11] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in MPI-IO," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, 2002.

[12] Y. Tsujita, H. Muguruma, K. Yoshinaga, A. Hori, M. Namiki, and Y. Ishikawa, "Improving collective i/o performance using pipelined two-phase i/o," in *Proceedings of the 2012 Symposium on High Performance Computing*, 2012.

[13] Y. Tsujita, K. Yoshinaga, A. Hori, M. Sato, M. Namiki, and Y. Ishikawa, "Multithreaded two-phase I/O: Improving collective MPI-IO performance on a lustre file system," in *Proceedings of the Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 232–235, 2014.

[14] W. Liu, M. Gerndt, and B. Gong, *Model-Based MPI-IO Tuning with Periscope Tuning Framework*, John Wiley and Sons Ltd, 2016.

[15] R. Filgueira, D. E. Singh, J. C. Pichel, F. Isaila, and J. Carretero, "Data locality aware strategy for two-phase collective," in *Proceedings of the High Performance Computing for Computational Science-VECPAR 2008*, pp. 137–149, 2008.

[16] H. Bui, J. Leigh, E. Jungy, V. Vishwanathy, and M. E. Papka, "Improving Data Movement Performance for Sparse Data Patterns on the Blue Gene/Q Supercomputer," in *Proceedings of the International Conference on Parallel Processing Workshops*, pp. 302–311, 2015.

[17] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Proceedings of the High PERFORMANCE Computing, Networking, Storage and Analysis*, 2011.

[18] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers," in *Proceedings of the IEEE International Conference on CLUSTER Computing*, pp. 70–80, 2017.

[19] W. keng Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 313–344, 2008.

[20] F. Broquedis, J. Clet-Ortega, S. Moreaud et al., "Hwloc: a generic framework for managing hardware affinities in HPC applications," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '10)*, pp. 180–186, 2010.

[21] A. Bermúdez, R. Casado, F. J. Quiles, T. M. Pinkston, and J. Duato, "Evaluation of a subnet management mechanism for InfiniBand networks," in *Proceedings of the 2003 International Conference on Parallel Processing, ICPP 2003*, pp. 117–124, 2003.

[22] S. S. Blackman, "Multiple target tracking with radar applications," *Dedham Ma Artech House Inc P*, vol. 1, pp. 204-205, 1986.

[23] R. Jonker and A. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, no. 4, pp. 325–340, 1987.

[24] G. Carpaneto, S. Martello, and P. Toth, "Algorithms and codes for the assignment problem," *Annals of Operations Research*, vol. 13, no. 1-4, pp. 193–223, 1988.

[25] P. Wong and R. F. V. der Wijngaart, "NAS Parallel Benchmarks I/O Version 2.4," 2003.